ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΗΛΕΚΤΡΟΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ ΚΑΙ ΔΙΚΤΥΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Δομές δεδομένων σε μνήμη φλας

Ιωάννης Α. Δημητρόπουλος

Επιβλέποντες: Παναγιώτης Μποζάνης, Επίκουρος Καθηγητής
Δημήτριος Κατσαρός, Συμβασιούχος ΠΔ407/80

ΒΟΛΟΣ
ΙΟΥΛΙΟΣ 2008

Περιεχόμενα

1 Εισαγωγικά – Βασικές έννοιες 7
1.1 Εισαγωγή……………………………………………………………………..8
1.2 Χαρακτηριστικά της μνήμης φλας ……………………………………………...8
1.3 Σύγκριση της NOR και της NAND μνήμης φλας ……………………………..11
Βιβλιογραφία κεφαλαίου……………………………………………………………...14

2 Η μνήμη φλας σαν συσκευή μπλοκ 15
2.1 Εισαγωγή..16
2.2 Η ιδέα της αντιστοίχισης μπλοκ (The Block-Mapping Idea)............................17
2.3 Δομές δεδομένων για αντιστοίχιση (Data Structure for Mapping)...................18
Βιβλιογραφία κεφαλαίου..28

3 Συστήματα αρχείων ειδικά για μνήμη φλας 29
3.1 Εισαγωγή..30
3.2 Συστήματα αρχείων με δομή Log (Ημερολογίου)
(Log-Structured File Systems)...31
3.3 Journaling Flash File System (JFFS)..34
3.4 YAFFS: Yet Another Flash File System..38
3.5 Trimble File System..41
3.6 Research-In-Motion File System...42
Βιβλιογραφία κεφαλαίου..47

4 Δομές δεδομένων ευρετηρίου (Indexing data structures) 44
4.1 Εισαγωγή..46
4.2 Το στρώμα BFTL..47
4.3 B^+ - δένδρα...64
4.4 μ - δένδρα...67
4.5 FlashDB...78
Βιβλιογραφία κεφαλαίου..88

5 Δομές δεδομένων για γεωγραφικά δεδομένα 89
5.1 Εισαγωγή..90
5.2 R–δένδρα...90
5.3 Αποδοτική υλοποίηση R – δένδρων σε μνήμη φλας...98
Βιβλιογραφία κεφαλαίου..105

6 Μηχανισμός εντοπισμού «καυτών» (hot) δεδομένων 107
6.1 Εισαγωγή..107
6.2 Μηχανισμός πολλαπλών συναρτήσεων κατακερματισμού.............................107
Βιβλιογραφία κεφαλαίου..110

7 Μηχανισμοί συλλογής απορριμμάτων και εξισορρόπησης φθοράς 111
7.1 Εισαγωγή..112
7.2 Εξισορρόπηση φθοράς βασισμένη στην ομαδοποίηση
(Group-based wear leveling)..113
7.3 Αποδοτικός μηχανισμός στατικής εξισορρόπησης φθοράς............................118

7.4 Μηχανισμός συλλογής απορριμμάτων πραγματικού χρόνου
(Real-time garbage collection mechanism)..122
Βιβλιογραφία κεφαλαίου..130

Συνολική βιβλιογραφία **131**

Πρόλογος

Η παρούσα διπλωματική συνιστά μία ενδελεχή εισαγωγή σε δομές δεδομένων και αλγορίθμους για **μνήμη φλας**. Κατεβλήθη, δε, προσπάθεια τόσο για την θεωρητική όσο και την πρακτική παρουσίαση των διαφόρων ζητημάτων. Πιο συγκεκριμένα, υιοθετήθηκε η προσέγγιση της περιγραφής των εκάστοτε αλγορίθμων σε μία «ψευδογλώσσα», ενώ δόθηκε έμφαση στην ανάλυση της πολυπλοκότητας των διαφόρων δομών δεδομένων, με την κατάλληλη μαθηματική αφαίρεση και τα ανάλογα μαθηματικά εργαλεία. Επιπρόσθετα, πολλές δομές δεδομένων, παρουσιάζονται πρακτικά, μέσω παραδειγμάτων και σχεδιαγραμμάτων, γεγονός που καθιστά ευκολότερη την κατανόησή τους από τον αναγνώστη.

Όσον αφορά την διάρθρωση της ύλης, πρέπει να σημειωθούν τα εξής: Στο πρώτο κεφάλαιο, παρουσιάζονται τα βασικά χαρακτηριστικά της μνήμης φλας καθώς και οι βασικές της κατηγορίες. Το δεύτερο κεφάλαιο αποτελεί μία εισαγωγή στις τεχνικές αντιστοίχισης μπλοκ σε φυσικές διευθύνσεις της μνήμης φλας, ενώ παράλληλα παρουσιάζονται τα δημοφιλή στρώματα μετάφρασης FTL (Flash Translation Layer) και NFTL (NAND Flash Translation Layer), τα οποία είναι κατάλληλα για μνήμη φλας. Στο τρίτο κεφάλαιο, παρουσιάζονται λεπτομερώς ορισμένα δημοφιλή συστήματα αρχείων ειδικά για μνήμη φλας, όπως για παράδειγμα το JFFS (Journalling Flash File System) και το YAFFS (Yet Another Flash File System). Στο τέτεταρτο κεφάλαιο, εξετάζονται διάφορες δομές δεδομένων ευρετηρίου, όπως για παράδειγμα τα B-δένδρα και τα μ-δένδρα. Παρουσιάζονται οι βασικές πράξεις (εισαγωγή, διαγραφή, αναζήτηση στοιχείου) πάνω σε αυτές τις δενδρικές δομές δεδομένων, με τη βοήθεια σχημάτων, καθώς και οι πολυπλοκότητες αυτών των δομών.

Στη συνέχεια της ύλης και συγκεκριμένα στο κεφάλαιο πέντε, εξετάζονται οι δομές δεδομένων σχετικά με την αποθήκευση γεωγραφικών/χωρικών δεδομένων, καθώς και η βελτιστοποίηση αυτών των δομών έτσι ώστε να είναι κατάλληλες προς χρήση σε μνήμη φλας. Στο κεφάλαιο έξι, παρουσιάζεται ένας μηχανισμός εντοπισμού «καυτών» δεδομένων (δεδομένα που ανανεώνονται τακτικά), ο οποίος αποτελείται από ένα σύνολο συναρτήσεων κατακερματισμού (hash functions). Τέλος, στο κεφάλαιο επτά, εξετάζονται διάφοροι μηχανισμοί συλλογής απορριμμάτων (garbage collection mechanism) και εξισορρόπησης φθοράς (wear leveling).

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Δομές δεδομένων και αλγόριθμοι.

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Μνήμη φλας, τεχνικές αντιστοίχισης μπλοκ, συστήματα αρχείων ειδικά για μνήμη φλας, δομές δεδομένων ευρετηρίου, συλλογή απορριμμάτων, εξισορρόπηση φθοράς.

Κεφάλαιο 1

Εισαγωγικά – Βασικές έννοιες

Περιεχόμενα

1.1 Εισαγωγή…………………………………………………………........... 8

1.2 Χαρακτηριστικά της μνήμης φλας…………………………………… 8

1.2.1 NOR μνήμη φλας……………………………………………………... 9

1.2.2 NAND μνήμη φλας…………………………………………………… 9

1.3 Σύγκριση της NOR και της NAND μνήμης φλας………………….. 11

1.3.1 Σύγκριση βασικών χαρακτηριστικών………………………………….. 11

1.3.2 Αρχιτεκτονική NAND και NOR μνήμης φλας……………………….... 11

1.3.3 Επιλέγοντας ανάμεσα σε NAND και NOR μνήμη φλας…………..…... 13

Βιβλιογραφία κεφαλαίου..14

1.1 Εισαγωγή

Τα τελευταία χρόνια, παρατηρείται μεγάλη εισροή τεχνολογικών γκάτζετ (gadget) στη καθημερινότητα του ανθρώπου, εξαιτίας κυρίως της επανάστασης που σημειώνεται στο τομέα των τηλεπικοινωνιών. Συσκευές παλάμης (hand-held devices), κινητά τηλέφωνα , ψηφιακές κάμερες, κινητές συσκευές αναπαραγωγής ήχου (portable music players) καθώς και πολλά άλλα συστήματα υπολογιστών έχουν γίνει αναπόσπαστο κομμάτι της ζωής των ανθρώπων, ενώ παράλληλα αυξάνονται και οι εφαρμογές για αυτές τις πλατφόρμες καθώς και η πολυπλοκότητα αυτών των προγραμμάτων. Επιπρόσθετα, η τεχνολογική πρόοδος που σημειώνεται στα **ενσωματωμένα συστήματα (embedded systems)** καθώς και στα **δίκτυα αισθητήρων (sensor networks),** βασίζεται κυρίως σε **συσκευές μικρής κλίμακας (small-scale devices),** στις οποίες οι **μαγνητικοί δίσκοι (magnetic disks)** είναι ακατάλληλοι. Ως αποτέλεσμα των παραπάνω, κρίνεται αναγκαίος ο σχεδιασμός αλγορίθμων και **δομών δεδομένων (data structures)** χαμηλής πολυπλοκότητας με μικρές απαιτήσεις σε μνήμη.

Ένα από τα πιο συνηθισμένα προβλήματα στο χώρο των υπολογιστών είναι η αποθήκευση μεγάλου όγκου πληροφορίας καθώς και η αποδοτική **αναζήτηση (searching)** και **ανάκτηση (retrieving)** αυτής της πληροφορίας. Υπάρχουν πολλοί αλγόριθμοι και δομές δεδομένων εσωτερικής μνήμης (internal memory), όπως δυαδικά δένδρα αναζήτησης (binary search trees), συναρτήσεις κατακερματισμού (hash functions), ψηφιακά δένδρα (tries), που αντιμετωπίζουν το παραπάνω πρόβλημα. Όταν όμως ο όγκος των δεδομένων είναι τόσο μεγάλος ώστε να μην χωράει στη κύρια μνήμη (main memory) του συστήματος, η ανάγκη για χρήση αλγορίθμων εξωτερικής μνήμης κρίνεται επιτακτική. Αυτά τα δεδομένα αποθηκεύονται σε **εξωτερικά συστήματα αποθήκευσης (external storage systems)** και προσπελαύνονται από κατάλληλους αλγορίθμους πραγματοποιώντας έτσι επιθυμητές λειτουργίες. Πολλές από τις δομές δεδομένων εσωτερικής μνήμης έχουν πλέον επεκταθεί και για χρήση σε εξωτερική μνήμη.

Το παραπάνω πρόβλημα όμως δεν έχει μελετηθεί αρκετά για την περίπτωση κινητών συσκευών με πολύ μικρή εσωτερική μνήμη και χαμηλή επεξεργαστική ισχύ. Εξαιτίας της μικρής ποσότητας εσωτερικής μνήμης, αυτές οι συσκευές δε μπορούν να διαχειριστούν μεγάλο όγκο πληροφορίας με αποτέλεσμα να κρίνεται αναγκαία η χρήση εξωτερικής μνήμης. Στις περισσότερες συσκευές παλάμης, η **μνήμη φλας (flash memory)** χρησιμοποιείται σαν εξωτερική μνήμη, εξαιτίας κυρίως της χαμηλής της τιμής και του συνεχώς αυξανόμενου μεγέθους της. Τα τελευταία χρόνια έχει γίνει αρκετή δουλεία πάνω σε συστήματα αρχείων ειδικά για μνήμη φλας (flash memory specific file systems), ενώ παράλληλα κάποιες κλασσικές δομές δεδομένων εξωτερικής μνήμης, όπως B-δένδρα και R-δένδρα, έχουν τροποποιηθεί για να είναι συμβατές με μνήμη φλας.

1.2 Χαρακτηριστικά της μνήμης φλας

Η μνήμη φλας είναι ένας τύπος **EEPROM (Electronically Erasable Programmable Read-Only Memory)** μνήμης. Είναι **μη ευμετάβλητη (non-volatile)** (διατηρεί τα δεδομένα της και χωρίς ρεύμα), με αποτέλεσμα να χρησιμοποιείται για τη μόνιμη αποθήκευση δεδομένων σε συσκευές παλάμης και κινητά τηλέφωνα. Παρουσιάζει

αντοχή σε δονήσεις (shock-resistant) ενώ διακρίνεται για τις **επιδόσεις της σε κατανάλωση ενέργειας (energy-efficient)**. Έχει μία μοναδική συμπεριφορά ανάγνωσης/εγγραφής/διαγραφής σε σύγκριση με τις υπόλοιπες μνήμες. Ο αριθμός των εγγραφών (writes) που πραγματοποιούνται σε μία μνήμη φλας είναι συγκεκριμένος. Ποικίλλει από 10.000 έως 1.000.000, ενώ όταν ξεπεραστούν αυτά τα όρια, η μνήμη φλας **φθείρεται (wears out)** και πρακτικά δε μπορεί να χρησιμοποιηθεί περαιτέρω.

Υπάρχουν δύο βασικές κατηγορίες μνήμης φλας. Η **NOR μνήμη φλας** και η **NAND μνήμη φλας**. Και στους δύο αυτούς τύπους, οι λειτουργίες εγγραφής, πρώτα «καθαρίζουν» τα bits (clear bits) (αλλάζουν την τιμή τους από 1 σε 0). Ο μόνος τρόπος να «θέσουμε» bits (set bits) (αλλαγή της τιμής από 0 σε 1) είναι να σβήσουμε μία ολόκληρη περιοχή μνήμης. Αυτή η περιοχή μνήμης έχει καθορισμένο μέγεθος (από αρκετά kilobytes μέχρι χιλιάδες kilobytes) και ονομάζεται **μονάδα σβησίματος (erase unit)**.

Εξαιτίας των παραπάνω χαρακτηριστικών, ειδικές δομές δεδομένων και αλγόριθμοι είναι απαραίτητοι για την αποδοτική χρήση της μνήμης φλας. Αυτοί οι αλγόριθμοι και οι δομές δεδομένων υποστηρίζουν **ανανέωση δεδομένων σε διαφορετική θέση (not-in-place updates of data)**, μειώνουν τον αριθμό των διαγραφών και **εξισορροπούν τη φθορά (wear leveling)** των μπλοκ της μνήμης.

1.2.1 NOR μνήμη φλας

Η NOR μνήμη φλας είναι η παλαιότερη από τους δύο τύπους και παρέχει **τυχαία προσπέλαση (random access).** Διευθυνσιοδοτείται σε επίπεδο byte από την Κεντρική Μονάδα Επεξεργασίας (CPU) και αν χρειαστεί μπορεί να χρησιμοποιηθεί σαν κύρια μνήμη RAM. Κυρίως χρησιμοποιείται για την αποθήκευση στατικών δεδομένων (static data) μίας και παρουσιάζει υψηλούς χρόνους εγγραφής. Το κύριο πλεονέκτημα της NOR μνήμης φλας είναι το ότι υποστηρίζει εγγραφή δεδομένων (writes) σε επίπεδο byte σε αντίθεση με τη NAND μνήμη φλας που υποστηρίζει εγγραφή δεδομένων σε επίπεδο σελίδας (page). Επιπλέον, η NOR μνήμη φλας, παρέχει ταχύτερους χρόνους προσπέλασης δεδομένων (περίπου 200ns) σε σύγκριση με την NAND μνήμη φλας (50-80μs), αλλά υστερεί σε όλα τα άλλα χαρακτηριστικά όπως η **πυκνότητα (density)** και η **αποδοτικότητα ισχύος (power efficiency)**.

1.2.2 NAND μνήμη φλας

Ένα τσιπάκι (chip) NAND μνήμης φλας αποτελείται από ένα καθορισμένο αριθμό **μπλοκ (blocks)**, όπου κάθε μπλοκ αποτελείται από ένα αριθμό από **σελίδες (pages)**. Κάθε σελίδα με τη σειρά της αποτελείται από μία περιοχή που περιέχει **κύρια δεδομένα (main data area)** και μία **ελεύθερη περιοχή (spare area)**. Η ελεύθερη περιοχή κάθε σελίδας χρησιμοποιείται για την αποθήκευση **κώδικα διόρθωσης λαθών (error correction code-ECC)** καθώς και άλλων πληροφοριών διαχείρισης. Κάθε μπλοκ μίας τυπικής NAND μνήμης φλας, μπορεί να σβηστεί (erased) έως 10^6 φορές. Όταν ξεπεραστεί αυτό το όριο, το μπλοκ φθείρεται (worns out) και υποφέρει από λάθη κατά την εγγραφή (write errors).

Καθώς η τεχνολογία των NAND μνημών φλας προοδεύει, έχουν παρουσιαστεί διαφορετικοί τύποι NAND μνήμης φλας. Η πρώτη «μικρού μπλοκ», **Μονού-Επιπέδου Κελιού (Single-Level Cell) SLC** μνήμη NAND (small block SLC NAND), είναι οργανωμένη έτσι ώστε κάθε μπλοκ να περιέχει 32 σελίδες και το μέγεθος κάθε σελίδας να είναι (512+16) bytes συμπεριλαμβανομένων και των 16 bytes της ελεύθερης περιοχής (spare area). Η επόμενη γενιά NAND μνήμης φλας ονομάζεται «μεγάλου μπλοκ» SLC NAND (large block SLC NAND) και παρέχει μεγαλύτερη χωρητικότητα. Πιο συγκεκριμένα, ο αριθμός σελίδων σε ένα μπλοκ στην «μεγάλου μπλοκ» μνήμη είναι διπλάσιος σε σχέση με αυτόν της «μικρού μπλοκ» και το μέγεθος της σελίδας είναι κατά τέσσερις φορές μεγαλύτερο.

Πρόσφατα εμφανίστηκε η **Πολλαπλού-Επιπέδου Κελιού (Multiple-Level Cell) MLC NAND** στην οποία κάθε τρανζίστορ μπορεί να βρίσκεται σε μία από τέσσερις διαφορετικές καταστάσεις δίνοντας τη δυνατότητα να κωδικοποιούμε δεδομένα έτσι ώστε να επιτυγχάνουμε πυκνότητα αποθήκευσης δύο bits ανά κελί μνήμης (memory cell), γεγονός που διπλασιάζει τη χωρητικότητα της NAND μνήμης φλας. Στη MLC NAND, κάθε σελίδα αποθηκεύει 4096 bytes δεδομένων (main data) ενώ το μέγεθος της ελεύθερης περιοχής (spare area) είναι 128 bytes. Ο αριθμός των σελίδων που περιέχονται σε ένα block ανέρχονται σε 128.

Επειδή η μνήμη φλας είναι **μονής εγγραφής (write-once)**, τα νέα δεδομένα δεν πανωγράφουν (overwrite) τα παλαιά σε κάθε διαδικασία ανανέωσης (update). Αντιθέτως, τα νέα δεδομένα γράφονται στον ελεύθερο/διαθέσιμο χώρο (free space) και οι παλαιές εκδόσεις των δεδομένων κηρύσσονται ως μη έγκυρες (invalidated). Η παραπάνω στρατηγική ανανέωσης ονομάζεται **ανανέωση εκτός θέσης (outplace update)**. Με άλλα λόγια, μία σελίδα NAND μνήμης φλας, πρέπει πρώτα να σβηστεί προτού τα νέα δεδομένα (updated data) γραφτούν σε αυτή (και στην ίδια ακριβώς θέση). Οι λειτουργίες διαγραφής πραγματοποιούνται σε επίπεδο μπλοκ, του οποίου το μέγεθος είναι κατά 64 ή 128 φορές μεγαλύτερο από αυτό της σελίδας. Στην «μεγάλου μπλοκ» SLC NAND, η εγγραφή δεδομένων σε μικρότερα τμήματα μέσα στη σελίδα είναι εφικτή κάνοντας χρήση του **τμηματικού προγραμματισμού σελίδας (partial page programming)**. Η ένδειξη NOP στον Πίνακα 1.1 που ακολουθεί, αναφέρεται στο μέγιστο αριθμό τμηματικού προγραμματισμού μίας σελίδας. Αξίζει να παρατηρήσουμε ότι ο NOP της MLC NAND έχει τιμή 1, γεγονός που μαρτυρά ότι η MLC NAND δεν υποστηρίζει τμηματικό προγραμματισμό σελίδας. Επιπρόσθετα, σύμφωνα με τον Πίνακα 1.1, οι λειτουργίες εγγραφής δεδομένων απαιτούν σχετικά μεγαλύτερους χρόνους σε σχέση με αυτές της ανάγνωσης δεδομένων. Ο λόγος της καθυστέρησης εγγραφής (write latency) προς την καθυστέρηση ανάγνωσης (read latency) είναι περίπου 3:3:1 στην «μεγάλου μπλοκ» SLC NAND, ενώ ο λόγος αυξάνεται σε 5:5:1 στην MLC NAND.

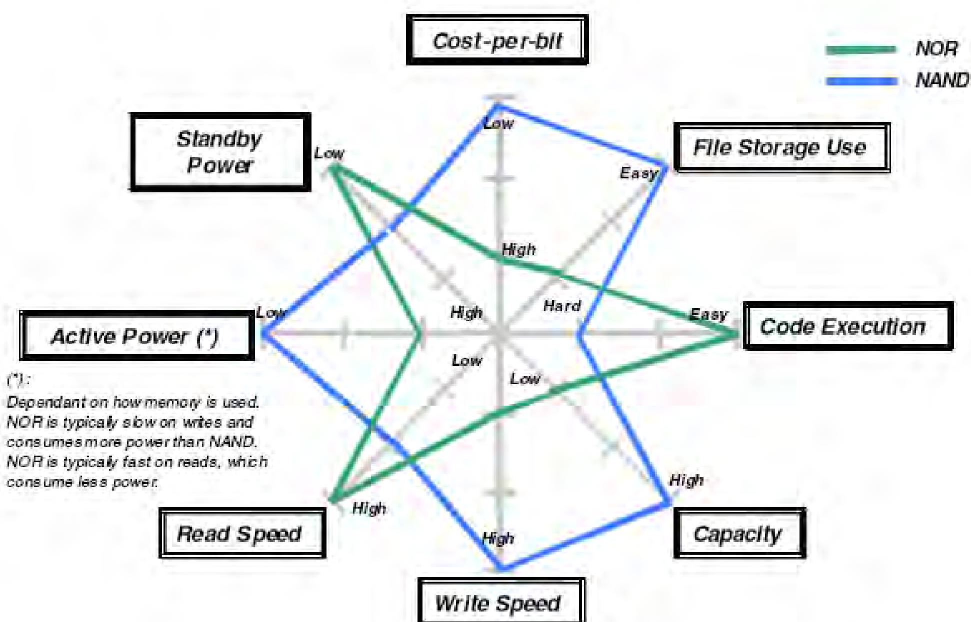
	SLC NAND (large block)	MLC NAND
page size	(2K+64)B	(4K+128)B
block size	(128+4)KB	(512+16)KB
# pages / block	64	128
NOP	4	1
read latency	77.8μs (2KB)	165.6μs (4KB)
write latency	252.8μs (2KB)	905.8μs (4KB)
erase latency	1500μs (128KB)	1500μs (512KB)

Πίνακας 1.1: Χαρακτηριστικά της SLC και της MLC NAND μνήμης φλας.

1.3 Σύγκριση της NOR και της NAND μνήμης φλας

1.3.1 Σύγκριση βασικών χαρακτηριστικών

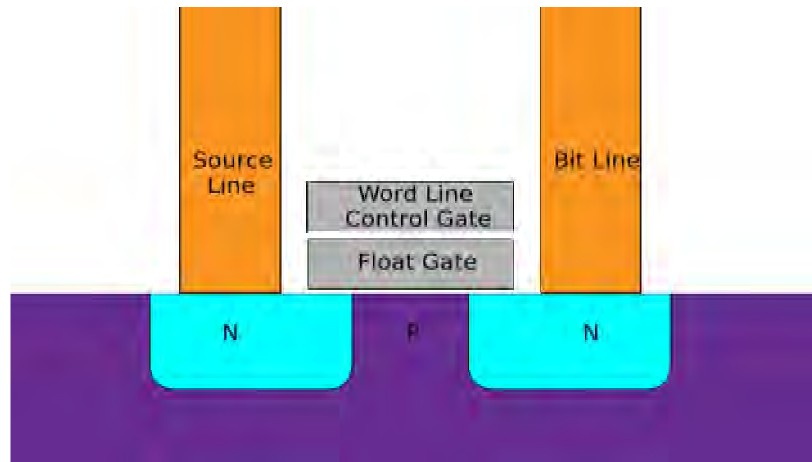
Το σχήμα 1.1 που ακολουθεί, παρουσιάζει συνοπτικά τις κυριότερες διαφορές των δύο τύπων μνήμης φλας πάνω σε βασικά σχεδιαστικά χαρακτηριστικά όπως: χωρητικότητα (capacity), ταχύτητα ανάγνωσης (read speed), ταχύτητα εγγραφής (write speed), κατανάλωση ισχύος σε ενεργή κατάσταση (active power consumption), κατανάλωση ισχύος σε κατάσταση αναμονής (standby power consumption), κόστος ανά bit (cost-per-bit) καθώς και ευχρηστία στην αποθήκευση αρχείων καθώς και κώδικα εφαρμογών (ease of use for file storage and code storage applications).



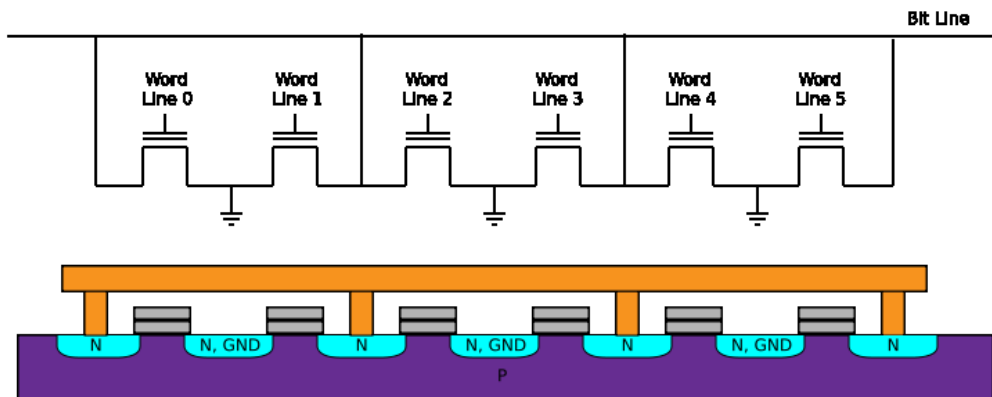
Σχήμα 1.1: Σύγκριση χαρακτηριστικών NOR και NAND μνήμης φλας.

1.3.2 Αρχιτεκτονική NAND και NOR μνήμης φλας

Στο εσωτερικό κύκλωμα της μνήμης NOR, τα ξεχωριστά κελιά μνήμης είναι συνδεδεμένα μεταξύ τους παράλληλα, γεγονός που δίνει τη δυνατότητα στη συσκευή μνήμης να επιτυγχάνει τυχαία προσπέλαση (random access). Η NOR μνήμη φλας είναι κατάλληλη για χαμηλής πυκνότητας (lower density), με υψηλές ταχύτητες ανάγνωσης, εφαρμογές. Τέτοιου είδους εφαρμογές ονομάζονται συνήθως **εφαρμογές αποθήκευσης κώδικα (code-storage applications)**.

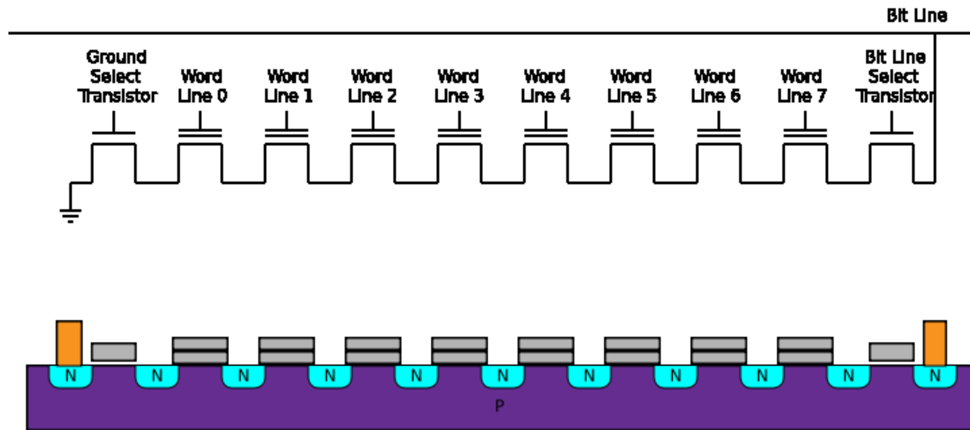


Σχήμα 1.2: Δομή ενός κελιού μνήμης φλας.



Σχήμα 1.3: Διάγραμμα συνδεσμολογίας NOR μνήμης φλας.

Η NAND μνήμη φλας είναι σχεδιασμένη για αποθήκευση δεδομένων υψηλής πυκνότητας. Το μέγεθος του τσιπ καθώς και το κόστος ανά bit (cost-per-bit) είναι μικρότερο, συγκριτικά με αυτό της NOR μνήμης φλας. Κάτι τέτοιο επιτεύχθηκε με την κατασκευή μίας διάταξης από οχτώ τρανζίστορ μνήμης συνδεδεμένα σε σειρά. Η NAND μνήμη φλας είναι κατάλληλη για χαμηλού κόστους (low-cost), υψηλής πυκνότητας (high-density), με υψηλές ταχύτητες διαγραφής/σβησίματος, εφαρμογές. Τέτοιου είδους εφαρμογές ονομάζονται συνήθως **εφαρμογές αποθήκευσης δεδομένων(data-storage applications)**.



Σχήμα 1.4: Διάγραμμα συνδεσμολογίας NAND μνήμης φλας.

1.3.3 Επιλέγοντας ανάμεσα σε NAND και NOR μνήμη φλας

Η απάντηση στο ερώτημα πότε να χρησιμοποιήσει κανείς NAND και πότε NOR μνήμη φλας εξαρτάται από τις απαιτήσεις του εκάστοτε συστήματος. Στον πίνακα 1.2 που ακολουθεί, παρουσιάζεται μία σύγκριση των επιδόσεων της μνήμης NAND και NOR.

	SLC NAND Flash (x8)	MLC NAND Flash (x8)	MLC NOR Flash (x16)
Density	512 Mbits[1] – 4 Gbits[2]	1Gbit to 16Gbit	16Mbit to 1Gbit
Read Speed	24 MB/s[3]	18.6 MB/s	103MB/s
Write Speed	8.0 MB/s	2.4 MB/s	0.47 MB/s
Erase Time	2.0 mSec	2.0mSec	900mSec
Interface	I/O – indirect access	I/O – indirect access	Random access
Application	Program/Data mass storage	Program/Data mass storage	eXecuteInPlace

Πίνακας 1.2: Λειτουργικές προδιαγραφές της NAND και NOR μνήμης φλας.

Τα χαρακτηριστικά της NAND μνήμης φλας είναι: υψηλή πυκνότητα, μεσαία ταχύτητα ανάγνωσης, υψηλή ταχύτητα εγγραφής, υψηλή ταχύτητα διαγραφής και έμμεση προσπέλαση. Τα χαρακτηριστικά της NOR μνήμης φλας είναι: χαμηλότερη πυκνότητα, υψηλή ταχύτητα ανάγνωσης, χαμηλή ταχύτητα εγγραφής, χαμηλή ταχύτητα διαγραφής και τυχαία προσπέλαση.

Όταν το σύστημα εκτελεί κώδικά απευθείας από τη μνήμη φλας, ή όταν η καθυστέρηση ανάγνωσης είναι σημαντικός παράγοντας απόδοσης για το σύστημα, τότε η καλύτερη επιλογή είναι η NOR μνήμη φλας. Από την άλλη πλευρά, η υψηλότερη πυκνότητα της NAND μνήμης φλας καθώς και οι υψηλές ταχύτητες σβησίματος και προγραμματισμού της, την καθιστούν ιδανική για εφαρμογές αποθήκευσης (storage applications). Ενώ το κέρδος από τις υψηλές ταχύτητες

προγραμματισμού στις συσκευές φλας είναι προφανές, οι επιδόσεις στις λειτουργίες σβησίματος/διαγραφής είναι εξίσου σημαντικές παρόλο που κάτι τέτοιο δεν είναι τόσο προφανές. Η μνήμη φλας, σε αντίθεση με τα μαγνητικά συστήματα μνήμης (magnetic memory systems), όπως για παράδειγμα σκληροί δίσκοι, απαιτεί ένα ξεχωριστό βήμα σβησίματος, για να μετατρέψει όλα τα bits σε "1", προτού η συσκευή προγραμματιστεί. Η κατανάλωση ισχύος είναι ένα ακόμη σημαντικό κριτήριο για πολλές εφαρμογές. Για κάθε εφαρμογή που βασίζεται σε εντατικά γραψίματα (write-intensive application), η NAND μνήμη φλας είναι η κατάλληλη επιλογή μιάς και καταναλώνει πολύ λιγότερη ισχύ από τη μνήμη NOR.

Όταν ένα σύστημα, όπως ένα κινητό τηλέφωνο τελευταίας τεχνολογίας, απαιτεί τόσο εκτέλεση κώδικα όσο και μεγάλη χωρητικότητα για αποθήκευση δεδομένων, τότε οι σχεδιαστές πρέπει να σκεφτούν εναλλακτικές λύσεις όπως για παράδειγμα να χρησιμοποιήσουν ταυτόχρονα και τους δύο τύπους μνήμης φλας σε συνδυασμό με μία Ψεύδο Στατική RAM (Pseudo Static RAM - PSRAM), ή να χρησιμοποιήσουν NAND μνήμη φλας σε συνδυασμό με DRAM χαμηλής ισχύος στην οποία θα «τρέχει» ο κώδικας. Συμπερασματικά, η καταλληλότερη μνήμη φλας είναι αυτή που προσφέρει τις επιθυμητές λειτουργίες και την ανάλογη πυκνότητα κάθε φορά, σε συνδυασμό πάντα με το χαμηλότερο κόστος.

Βιβλιογραφία κεφαλαίου

[1] Eran Gal and Sivan Toledo. Algorithms and Data Structures for Flash Memories. (χρήση στις σελίδες 8-9)

[2] NAND vs. NOR Flash Memory. Technology Overview, Toshiba America Electronic Components. (χρήση στις σελίδες 11-14)

[3] www. wikipedia.org. (χρήση στις σελίδες 12-13)

[4] Dongwon Kang, Dawoon Jung, Jeong-Uk Kang and Jin-Soo Kim. μ - Tree: An Ordered Index Structure for NAND Flash Memory. (χρήση στις σελίδες 9-10)

Κεφάλαιο 2

Η μνήμη φλας σαν συσκευή μπλοκ

Περιεχόμενα

2.1 Εισαγωγή…………………………………………………………… 16

2.2 Η ιδέα της αντιστοίχισης μπλοκ (The Block – Mapping Idea)…. 17

2.3 Δομές δεδομένων για αντιστοίχιση

(Data structures for Mapping).. 18

2.3.1 Απευθείας πίνακας αντιστοίχισης (Direct map)

και Ανάστροφος πίνακας αντιστοίχισης (Inverse map)....................... 18

2.3.2 Flash Translation Layer (FTL)………………………………………. 20

2.3.2.1 Τα χαρακτηριστικά του FTL.. 20

2.3.2.2 Τεχνικές αντιστοίχισης μπλοκ του FTL…... 21

2.3.3 NAND Flash Translation Layer (NFTL)…………………….......... 23

2.3.3.1 Τεχνικές αντιστοίχισης μπλοκ του NFTL…………………………… 23

2.3.3.2 Τεχνικές βελτίωσης της απόδοσης του NFTL………………………. 25

2.3.3.2.1 Πίνακας αναζήτησης (Lookup Table)……………………………... 25

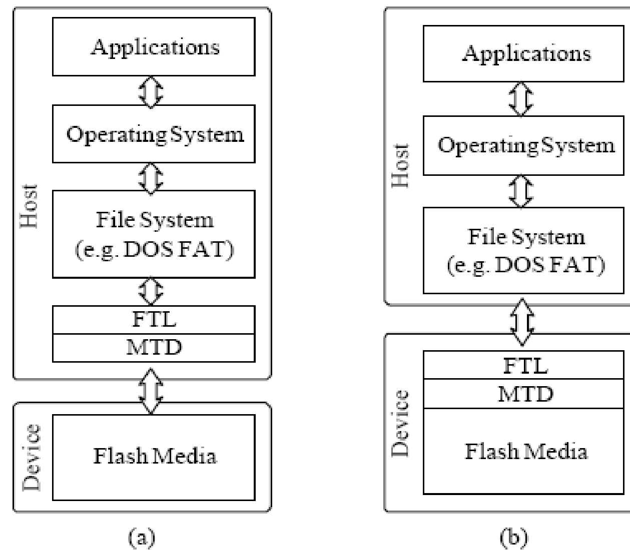
2.3.3.2.2 Κρυφή μνήμη σελίδας (Page Cache)………………………………. 26

2.3.4 Αντιστοίχιση μπλοκ με μεταβλητό μέγεθος………………………….. 27

Βιβλιογραφία κεφαλαίου..28

2.1 Εισαγωγή

Η μνήμη φλας, προσπελαύνεται από τα ενσωματωμένα συστήματα (embedded systems) είτε σαν ένα **ανεπεξέργαστο μέσο (raw medium),** είτε έμμεσα σαν μία **συσκευή προσανατολισμένη σε μπλοκ (block-oriented device)**. Πιο συγκεκριμένα, η διαχείριση της μνήμης φλας πραγματοποιείται είτε από το λογισμικό, σε ένα σύστημα host (σχήμα 2.1 (a)) (σαν ένα ακατέργαστο μέσο), είτε σε επίπεδο υλικού και κυκλωμάτων μέσα στην ίδια τη συσκευή φλας (σχήμα 2.1 (b)).



Σχήμα 2.1: Δύο τύποι προϊόντων μνήμης φλας.

Σε αυτό το κεφάλαιο θα μελετήσουμε τη μνήμη φλας σαν **συσκευή μπλοκ (block device)**. Η αντιμετώπιση της μήνης φλας σαν συσκευή μπλοκ, επιτρέπει σε μπλοκ δεδομένων καθορισμένου μεγέθους, να γραφτούν και να διαβαστούν όπως ακριβώς οι τομείς των δίσκων (disk sectors). Αυτό έχει ως αποτέλεσμα, η μνήμη φλας να μπορεί να χρησιμοποιεί **συστήματα αρχείων (file systems)** που είναι σχεδιασμένα για μαγνητικούς δίσκους όπως για παράδειγμα το **FAT (File Allocation Table)**. Ο κώδικας του συστήματος αρχείων, καλεί έναν οδηγό της συσκευής (device driver), αιτώντας λειτουργίες ανάγνωσης/εγγραφής μπλοκ. Ο οδηγός της συσκευής αποθηκεύει και ανακτά τα μπλοκ από τη μνήμη φλας.

Παρόλα αυτά, η **αντιστοίχιση των μπλοκ (block mapping)** σε διευθύνσεις της μνήμης φλας, με απλό και γραμμικό τρόπο, παρουσιάζει δύο βασικά προβλήματα. Πρώτον, κάποια μπλοκ δεδομένων μπορεί να γραφτούν πολύ περισσότερες φορές από κάποια άλλα. Η κατάσταση αυτή δεν αποτελεί πρόβλημα για τα παραδοσιακά συστήματα αρχείων με αποτέλεσμα να μην λαμβάνουν μέριμνα για τέτοιου είδους περιπτώσεις. Όταν όμως το παραδοσιακό σύστημα αρχείων χρησιμοποιείται σε μνήμη φλας, οι συχνά χρησιμοποιούμενες **μονάδες σβησίματος (erase units)** φθείρονται (wear out) γρήγορα, οι χρόνοι προσπέλασης σε αυτές αυξάνονται και τελικά, με το πέρασμα του χρόνου, καταστρέφονται. Το πρόβλημα αυτό, μπορεί να αντιμετωπιστεί αν χρησιμοποιηθεί ένας πιο εξειδικευμένος μηχανισμός αντιστοίχισης μπλοκ σε μνήμη φλας, καθώς και με τη μετακίνηση των μπλοκ. Οι τεχνικές που χρησιμοποιούν τέτοιες στρατηγικές ονομάζονται **τεχνικές εξισορρόπησης φθοράς (wear leveling techniques)**.

Το δεύτερο πρόβλημα, που προκύπτει από την απλή αντιστοίχιση των μπλοκ σε διευθύνσεις της μνήμης φλας, είναι η αδυναμία του να γράψουμε μπλοκ δεδομένων με μέγεθος μικρότερο από αυτό της μονάδας σβησίματος (erase unit). Ας υποθέσουμε για παράδειγμα ότι το μέγεθος των μπλοκ δεδομένων που χρησιμοποιεί το σύστημα αρχείων είναι 4 KB και το μέγεθος της μονάδας σβησίματος της μνήμης φλας είναι 128 KB. Εάν τα μπλοκ των 4 KB αντιστοιχίζονται σε διευθύνσεις της μνήμης φλας με βάση τον απλό τρόπο αντιστοίχισης, τότε η εγγραφή ενός μπλοκ 4 KB απαιτεί την αντιγραφή μίας 128 KB μονάδας σβησίματος στη μνήμη RAM, να πανωγράψουμε (overwrite) την κατάλληλη περιοχή μεγέθους 4 KB καθώς και να σβήσουμε τη μονάδα σβησίματος της μνήμης φλας και να την αντιγράψουμε από τη μνήμη RAM. Επιπρόσθετα, αν συμβεί διακοπή ρεύματος προτού η μονάδα σβησίματος αντιγραφεί ολόκληρη στη μνήμη φλας, τότε και τα 128KB δεδομένων χάνονται. Αντίθετα, στους μαγνητικούς δίσκους, μόνο το μπλοκ των 4 KB θα χανόταν. Η τεχνική εξισορρόπησης φθοράς (wear leveling technique) , αντιμετωπίζει και αυτή τη πρόκληση.

2.2 Η ιδέα της αντιστοίχισης μπλοκ (The Block-Mapping Idea)

Η βασική ιδέα όλων των τεχνικών εξισορρόπησης φθοράς (wear leveling techniques) είναι να αντιστοιχίσουν τον αριθμό του μπλοκ δεδομένων, ο οποίος ονομάζεται **εικονικός αριθμός μπλοκ (virtual block number)**, σε μία φυσική διεύθυνση στη μνήμη φλας, η οποία ονομάζεται **τομέας (sector)**. Όταν ένα εικονικό μπλοκ χρειάζεται να επανεγγραφεί/ανανεωθεί, τα νέα δεδομένα δεν πανωγράφουν τον τομέα στον οποίο είναι ήδη αποθηκευμένο το μπλοκ. Αντιθέτως, τα νέα δεδομένα γράφονται σε άλλο τομέα, η παλιά έκδοση των δεδομένων θεωρείται μη έγκυρη (invalid) και ο πίνακας αντιστοίχισης εικονικών μπλοκ σε τομείς (virtual-block-to-sector map) ανανεώνεται έτσι ώστε να περιέχει την τελευταία τροποποίηση. Η παραπάνω στρατηγική ανανέωσης δεδομένων (data update strategy), ονομάζεται **ανανέωση εκτός θέσης (out-place update)**. Με άλλα λόγια, τα δεδομένα που είναι αποθηκευμένα στη μνήμη φλας δε μπορούν να επανεγγραφούν/ανανεωθούν αν πρώτα δεν διαγραφούν.

Τυπικά, ο τομέας (sector) έχει καθορισμένο μέγεθος και καταλαμβάνει ένα μέρος της μονάδας σβησίματος. Στις NAND μνήμες φλας, ο τομέας καταλαμβάνει μία σελίδα (flash page). Στις NOR μνήμες φλας μπορεί να χρησιμοποιηθούν τομείς με μεταβλητό μήκος/μέγεθος.

Η παραπάνω τεχνική αντιστοίχισης εξυπηρετεί πολλούς σκοπούς:

- Πρώτον, η εγγραφή συχνά τροποποιημένων μπλοκ δεδομένων σε διαφορετικούς τομείς κάθε φορά, φθείρει ομοιόμορφα τις διάφορες μονάδες σβησίματος (erase units).
- Δεύτερον, η παραπάνω αντιστοίχιση επιτρέπει την εγγραφή ενός μπλοκ στη μνήμη φλας χωρίς να χρειάζεται να σβηστεί και να επανεγγραφεί μία ολόκληρη μονάδα σβησίματος.
- Τρίτον, επιτρέπει στα μπλοκ δεδομένων να γράφονται ατομικά. Πιο συγκεκριμένα, αν κατά τη διάρκεια μίας εγγραφής πραγματοποιηθεί διακοπή ρεύματος, όταν η μνήμη φλας επαναλειτουργήσει, το μπλοκ επανέρχεται στην κατάσταση που ήταν πριν την εγγραφή.

Η ατομικότητα επιτυγχάνεται χρησιμοποιώντας τη παρακάτω τεχνική. Κάθε τομέας σχετίζεται με μία μικρή κεφαλίδα (header), η οποία προσαρτάται είτε στον τομέα ή κάπου αλλού μέσα στη μονάδα σβησίματος. Όταν θέλουμε να γράψουμε ένα μπλοκ δεδομένων, το λογισμικό αναζητά ένα ελεύθερο (free) και σβησμένο (erased) τομέα. Σε αυτή την κατάσταση, όλα τα bits τόσο στον τομέα όσο και στην κεφαλίδα είναι 1. Στη συνέχεια, το **ελεύθερο/χρησιμοποιημένο bit (free/used bit)** λαμβάνει τιμή 0 γεγονός που υποδεικνύει ότι ο τομέας δεν είναι πια ελεύθερος. Αμέσως μετά, ο εικονικός αριθμός μπλοκ γράφεται στην κεφαλίδα και τα νέα δεδομένα εγγράφονται στον επιλεγμένο τομέα. Στη συνέχεια, το **prevalid/valid bit** λαμβάνει τιμή 0 γεγονός που υποδεικνύει ότι ο τομέας είναι έτοιμος για ανάγνωση. Τέλος, το **έγκυρο/απαρχαιωμένο bit (valid/obsolete bit)** στην κεφαλίδα του παλιού τομέα, λαμβάνει τιμή 0 υποδεικνύοντας ότι δεν περιέχει πια το πιο πρόσφατο αντίγραφο του εικονικού μπλοκ.

Σε ορισμένες περιπτώσεις, η παραπάνω διαδικασία μπορεί να βελτιστοποιηθεί συνδυάζοντας το **ελεύθερο/χρησιμοποιημένο bit (free/used bit)** με τον εικονικό αριθμό μπλοκ. Αν ο εικονικός αριθμός του μπλοκ αποτελείται μόνο από άσσους (1), τότε ο τομέας είναι ελεύθερος, διαφορετικά χρησιμοποιείται.

Εάν παρουσιαστεί διακοπή ρεύματος κατά τη διάρκεια μίας εγγραφής, τότε η μνήμη φλας μπορεί να βρεθεί σε δύο πιθανές καταστάσεις. Πιο συγκεκριμένα, εάν η διακοπή παρουσιαστεί πριν ο νέος τομέας σημειωθεί ως έγκυρος (valid), τα περιεχόμενά του αγνοούνται όταν η μνήμη φλας επαναχρησιμοποιηθεί, και το **έγκυρο/απαρχαιωμένο bit (valid/obsolete bit)** λαμβάνει τιμή 1 για να υποδείξει ότι ο τομέας είναι έτοιμος για σβήσιμο (erasure). Εάν όμως η διακοπή ρεύματος παρουσιαστεί αφού πρώτα ο νέος τομέας σημειωθεί ως έγκυρος και πριν ο παλαιός τομέας σημειωθεί ως απαρχαιωμένος (obsolete), τότε και οι δύο εκδόσεις θεωρούνται αποδεκτές με αποτέλεσμα το σύστημα να επιλέγει τη μία έκδοση και να σημειώνει την άλλη ως απαρχαιωμένη.

2.3 Δομές Δεδομένων για αντιστοίχιση (Data structures for Mapping)

2.3.1 Απευθείας πίνακας αντιστοίχισης (Direct map) και Ανάστροφος πίνακας αντιστοίχισης (Inverse map)

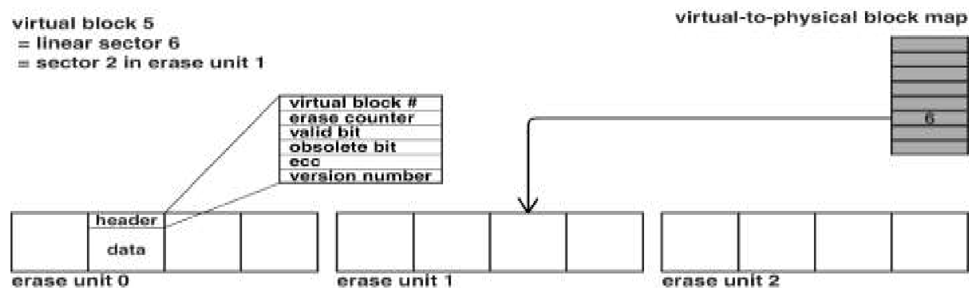
Μία από τις βασικότερες προκλήσεις στη μνήμη φλας είναι η εύρεση του τομέα που περιέχει ένα ζητούμενο μπλοκ δεδομένων. Βασικά, υπάρχουν δύο είδη δομών δεδομένων για τέτοιες αντιστοιχίσεις/απεικονίσεις. Ο **απευθείας πίνακας αντιστοίχισης (direct map)** είναι ένας πίνακας ο οποίος αποθηκεύει στην i θέση του το δείκτη/αριθμό (index) του τομέα που περιέχει το μπλοκ i. Ο **ανάστροφος πίνακας αντιστοίχισης (inverse map)** αποθηκεύει στην i θέση του το αναγνωριστικό του μπλοκ που είναι αποθηκευμένο στον τομέα i. Με άλλα λόγια, ο απευθείας πίνακας αντιστοίχισης, επιτρέπει αποδοτική απεικόνιση/αντιστοίχιση μπλοκ σε τομείς ενώ ο ανάστροφος πίνακας αντιστοίχισης επιτρέπει αποδοτική απεικόνιση/αντιστοίχιση τομέων σε μπλοκ. Σε ορισμένες περιπτώσεις ο απευθείας πίνακας αντιστοίχισης δεν είναι ένας απλός πίνακας αλλά μία πιο σύνθετη δομή δεδομένων. Όποια μορφή όμως

και αν έχει, ο απευθείας πίνακας αντιστοίχισης επιτυγχάνει αποδοτική αντιστοίχιση μπλοκ σε τομείς. Από την άλλη πλευρά, ο ανάστροφος πίνακας αντιστοίχισης, μπορεί να μην αποθηκεύεται σε συνεχόμενες θέσεις στη φυσική μνήμη (physical memory).

Ο ανάστροφος πίνακας αντιστοίχισης αποθηκεύεται στη συσκευή φλας. Όταν ένα μπλοκ γράφεται σε ένα τομέα, τότε και το αναγνωριστικό του μπλοκ γράφεται στην ίδια μονάδα σβησίματος με το μπλοκ έτσι ώστε να μπορούν να σβήνονται μαζί. Το αναγνωριστικό του μπλοκ, μπορεί να αποθηκευτεί σε μία κεφαλίδα αμέσως, προσπερνώντας τα δεδομένα, ή να αποθηκευτεί σε κάποια άλλη περιοχή της μονάδας σβησίματος, για παράδειγμα σε ένα τομέα αποκλειστικά για αποθήκευση αριθμών μπλοκ. Ο κύριος σκοπός του ανάστροφου πίνακα αντιστοίχισης είναι να ανακατασκευάζει έναν απευθείας πίνακα αντιστοίχισης κατά τη διάρκεια της αρχικοποίησης του συστήματος (όταν μία συσκευή φλας εισέρχεται στο σύστημα ή όταν το σύστημα εκκινεί-boot).

Ο απευθείας πίνακας αντιστοίχισης αποθηκεύεται τουλάχιστον τμηματικά στη μνήμη RAM. Ο λόγος για τον οποίο ο απευθείας πίνακας αντιστοίχισης αποθηκεύεται στη RAM είναι το ότι εξορισμού υποστηρίζει γρήγορες αναζητήσεις (fast lookups). Το γεγονός αυτό συνεπάγεται ότι, όταν ένα μπλοκ επανεγγράφεται και μετακινείται από έναν τομέα σε κάποιον άλλον, μία καθορισμένη θέση αναζήτησης (fixed lookup location) πρέπει να ανανεωθεί. Η μνήμη φλας όμως δεν υποστηρίζει τέτοιου είδους «εντός θέσης» τροποποιήσεις (in-place modifications).

Συνοψίζοντας, ο ανάστροφος πίνακας αντιστοίχισης που βρίσκεται στη μνήμη φλας, εξασφαλίζει ότι οι τομείς θα συσχετίζονται πάντα με τα μπλοκ που περιέχουν. Ο απευθείας πίνακας αντιστοίχισης που αποθηκεύεται στη μνήμη RAM, επιτρέπει στο σύστημα να βρίσκει γρήγορα τον τομέα που περιέχει το ζητούμενο μπλοκ. Αυτές οι δομές δεδομένων αντιστοίχισης, παρουσιάζονται και στο σχήμα 2.2.



Σχήμα 2.2: Αντιστοίχιση μπλοκ σε φλας συσκευή. Ο γκρι πίνακας πάνω δεξιά είναι ο απευθείας πίνακας αντιστοίχισης μπλοκ σε τομείς που βρίσκεται στη RAM. Κάθε τομέας περιέχει δεδομένα και μία κεφαλίδα. Η κεφαλίδα περιλαμβάνει τον δείκτη του εικονικού μπλοκ που είναι αποθηκευμένο στον τομέα, έναν μετρητή σβησίματος (erase counter), το έγκυρο/απαρχαιωμένο bit (valid/obsolete bit) και ίσως έναν κώδικα διόρθωσης λαθών (error-correction code) και έναν αριθμό έκδοσης (version number). Οι εικονικοί αριθμοί των μπλοκ στις κεφαλίδες των τομέων συνιστούν τον ανάστροφο πίνακα αντιστοίχισης από τον οποίο κατασκευάζεται και ο απευθείας πίνακας. Ο αριθμός έκδοσης, επιτρέπει στο σύστημα να αποφασίσει, ποιος από τους δύο έγκυρους τομείς που περιέχουν το ίδιο εικονικό μπλοκ, είναι ο πιο πρόσφατος.

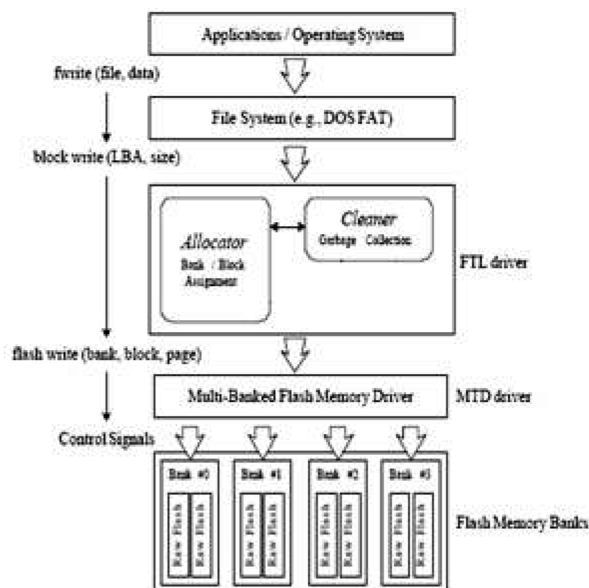
Ο απευθείας πίνακας αντιστοίχισης δεν είναι απολύτως απαραίτητος. Το σύστημα, μπορεί να ψάξει σειριακά κατά μήκος του ανάστροφου πίνακα αντιστοίχισης, για να βρει έναν έγκυρο τομέα ο οποίος περιέχει το ζητούμενο μπλοκ. Η διαδικασία αυτή είναι αργή αλλά πιο αποδοτική συγκριτικά με τη χρήση μνήμης RAM. Επιτρέποντας σε κάθε μπλοκ να αποθηκεύεται σε ένα μικρό αριθμό από τομείς, επιταχύνουμε τη

διαδικασία της αναζήτησης. Η παραπάνω τεχνική, που μοιάζει αρκετά με τις κρυφές μνήμες (cache memories), μειώνει το ποσό της μνήμης RAM που απαιτείται για τις αναζητήσεις αλλά ταυτόχρονα μειώνει την ευελιξία της απεικόνισης. Η μειωμένη ευελιξία μπορεί να οδηγήσει σε πιο συχνές διαγραφές και να επιταχύνει τη φθορά της μνήμης φλας.

2.3.2 Flash Translation Layer (FTL)

2.3.2.1 Χαρακτηριστικά του FTL

Η στρωματοποιημένη σχεδίαση (layered design) χρησιμοποιείται στην υλοποίηση φλας-συστημάτων αποθήκευσης (flash-memory storage systems). Τόσο ο οδηγός MTD (Memory Technology Device driver) όσο και ο οδηγός FTL (FTL driver) είναι δύο βασικά στρώματα για τη διαχείριση της μνήμης φλας, όπως φαίνεται και στο σχήμα που ακολουθεί.



Σχήμα 2.3: Μία τυπική αρχιτεκτονική συστήματος για συστήματα αποθήκευσης κατάλληλα για μνήμες φλας.

Ο οδηγός MTD παρέχει λειτουργίες χαμηλού επιπέδου όπως ανάγνωση (read), εγγραφή (write) και διαγραφή (erase). Βασισμένοι στις παραπάνω λειτουργίες του MTD, αλγόριθμοι υψηλού επιπέδου όπως **εξισορρόπησης φθοράς (wear leveling)**, **συλλογής απορριμμάτων (garbage collection)** και **μετάφρασης φυσικών/λογικών διευθύνσεων (physical/logical address translation)**, υλοποιούνται στον οδηγό του FTL. Ο σκοπός του οδηγού FTL είναι να παρέχει υπηρεσίες στις εφαρμογές των χρηστών και στο σύστημα αρχείων (file system), έτσι ώστε να προσπελαύνεται η μνήμη φλας σαν μία συσκευή προσανατολισμένη σε μπλοκ (block-oriented device).

Όπως φαίνεται και στο σχήμα 2.3, ο οδηγός FTL αποτελείται από δύο βασικά συστατικά: το **συστατικό ανάθεσης (allocator)** και το **συστατικό εκκαθάρισης**

(cleaner). Το συστατικό ανάθεσης είναι υπεύθυνο για την εύρεση σελίδων μνήμης φλας προς εγγραφή, καθώς και για την αντιστοίχιση λογικών διευθύνσεων των μπλοκ (Logical Block Addresses-LBA) σε φυσικές διευθύνσεις (Physical Block Addresses-PBA). Το συστατικό εκκαθάρισης είναι υπεύθυνο για την **ανάκτηση σελίδων (page reclamation)** που περιέχουν μη έγκυρα δεδομένα (invalid data). Η παραπάνω λειτουργία είναι ευρέως γνωστή ως **μηχανισμός συλλογής απορριμμάτων (garbage collection mechanism)**. Μία ακόμη βασική λειτουργία για τη διαχείριση της μνήμης φλας είναι η **εξισορρόπηση φθοράς (wear leveling)** η οποία διανέμει ομοιόμορφα των αριθμό των διαγραφών για κάθε μπλοκ (εξαιτίας βέβαια του περιορισμένου αριθμού διαγραφών για κάθε μπλοκ). Η κατάλληλη σχεδίαση τόσο συστατικού ανάθεσης (allocator) όσο και του συστατικού εκκαθάρισης (cleaner), όχι μόνο βελτιώνει την απόδοση του συστήματος αποθήκευσης (storage system) της μνήμης φλας αλλά αυξάνει και την διάρκεια ζωής της.

2.3.2.2 Τεχνικές αντιστοίχισης μπλοκ του FTL

Το Flash Translation Layer (FTL) παρέχει τεχνικές για την αποθήκευση ενός μέρους του απευθείας πίνακα αντιστοίχισης (direct map) μέσα στη μνήμη φλας μειώνοντας παράλληλα το κόστος της ανανέωσης του πίνακα μέσα στη συσκευή φλας. Το FTL χρησιμοποιεί ένα συνδυασμό τεχνικών για να επιτύχει αντιστοίχιση μπλοκ σε τομείς (block-to-sector mapping) :

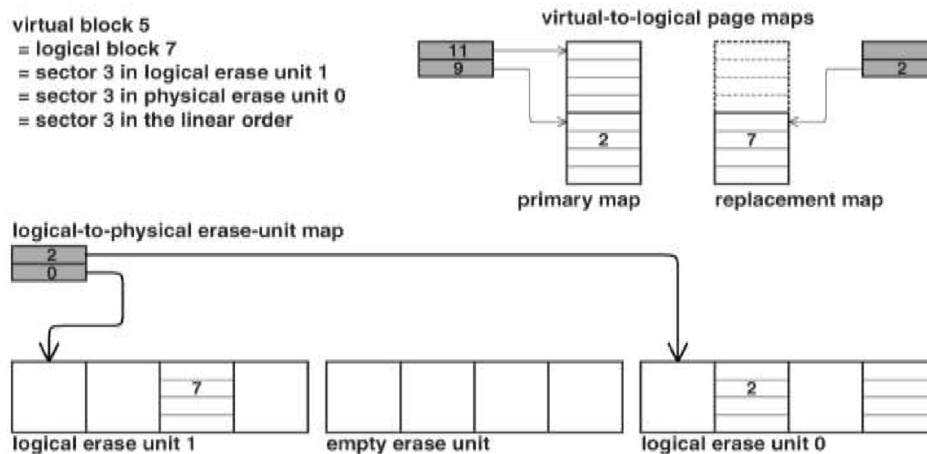
1 Ο αριθμός του μπλοκ αντιστοιχίζεται πρώτα σε ένα λογικό αριθμό μπλοκ (logical block number) ο οποίος αποτελείται από το λογικό αριθμό της μονάδας σβησίματος (logical erase unit number) (ο αριθμός αυτός καθορίζεται από τα πιο σημαντικά bits του λογικού αριθμού μπλοκ) καθώς και από ένα δείκτη του τομέα (sector index) που βρίσκεται μέσα στη μονάδα σβησίματος. Ο παραπάνω μηχανισμός επιτρέπει στους έγκυρους τομείς μίας μονάδας σβησίματος να αντιγράφονται σε πρόσφατα «σβησμένες» (newly-erased) μονάδες σβησίματος χωρίς να χρειάζονται αλλαγές στον πίνακα αντιστοίχισης μπλοκ σε λογικά μπλοκ (block-to-logical-block map), μιας και κάθε τομέας αντιγράφεται στην ίδια ακριβώς θέση στην νέα μονάδα σβησίματος.

2 Ο πίνακας αντιστοίχισης μπλοκ σε λογικά μπλοκ (block-to-logical-block map) αποθηκεύεται τμηματικά στη μνήμη RAM και στη μνήμη φλας. Η αντιστοίχιση των πρώτων μπλοκ αποθηκεύεται συνήθως στη μνήμη RAM ενώ των υπολοίπων στη μνήμη φλας.

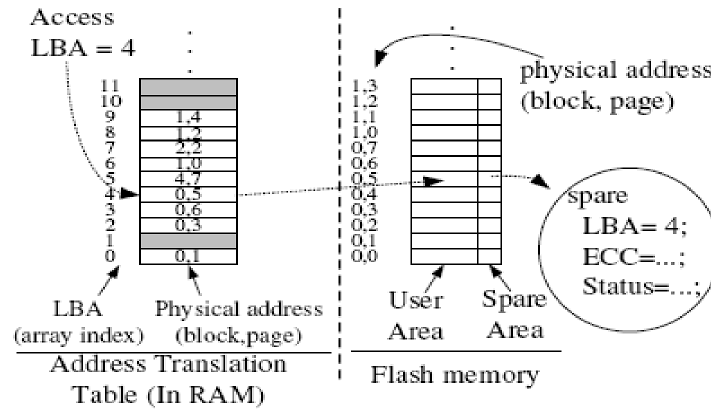
3 Το τμήμα του πίνακα αντιστοίχισης μπλοκ σε λογικά μπλοκ που βρίσκεται στη μνήμη φλας, δεν αποθηκεύεται σε συνεχόμενες θέσεις αλλά διασκορπίζεται μέσα στη συσκευή φλας μαζί με έναν ανάστροφο πίνακα αντιστοίχισης. Ένας απευθείας πίνακας αντιστοίχισης που βρίσκεται στη RAM, και ανακατασκευάζεται κατά την αρχικοποίηση του συστήματος, δείχνει στους τομείς του ανάστροφου πίνακα αντιστοίχισης . Για την εύρεση του λογικού αριθμού ενός μπλοκ, το σύστημα βρίσκει πρώτα τον τομέα που περιέχει την αντιστοίχιση, στον πίνακα κορυφαίου επιπέδου στη RAM (top-level RAM map), και στη συνέχεια ανακτά την αντιστοίχιση.

4 Όταν ένα μπλοκ επανεγγράφεται και μετακινείται σε νέο τομέα, η αντιστοίχιση του μπλοκ πρέπει να αλλάξει. Για να γίνει κάτι τέτοιο χωρίς να επανεγγράψουμε το σχετικό μπλοκ αντιστοίχισης, χρησιμοποιούμε έναν εφεδρικό πίνακα αντιστοίχισης ο οποίος αποθηκεύεται στη μνήμη φλας. Εάν η σχετική εγγραφή (relevant entry) στον εφεδρικό πίνακα αντιστοίχισης είναι διαθέσιμη (αποτελείται μόνο από 1), η αρχική εγγραφή (original entry) στον κύριο πίνακα αντιστοίχισης σβήνεται (όλα τα bits μετατρέπονται σε 0) και η νέα αντιστοίχιση γράφεται στον εφεδρικό πίνακα. Διαφορετικά, ο τομέας αντιστοίχισης πρέπει να επανεγγραφεί. Κατά τη διάρκεια μίας αναζήτησης, εάν η εγγραφή αντιστοίχισης αποτελείται μόνο από 0, τότε το σύστημα αναζητά την αντιστοίχιση στον εφεδρικό πίνακα. Ο εφεδρικός πίνακας αντιστοίχισης δεν περιέχει έναν εφεδρικό τομέα για κάθε τομέα αντιστοίχισης.

5 Τέλος, οι λογικές μονάδες σβησίματος (logical erase units) αντιστοιχίζονται σε φυσικές μονάδες σβησίματος (physical erase units) μέσω ενός μικρού πίνακα απευθείας αντιστοίχισης που βρίσκεται στη RAM. Εξαιτίας του μικρού του μεγέθους (μία εγγραφή ανά μονάδα σβησίματος, όχι ανά τομέα), η επιβάρυνση της μνήμης RAM είναι μικρή. Κατασκευάζεται κατά την αρχικοποίηση του συστήματος μέσω του ανάστροφου πίνακα αντιστοίχισης. Κάθε φυσική μονάδα σβησίματος αποθηκεύει το λογικό της αριθμό. Αυτός ο απευθείας πίνακας αντιστοίχισης ανανεώνεται κάθε φορά που ανακτάται μία μονάδα σβησίματος.



Σχήμα 2.4: Ένα παράδειγμα των δομών απεικόνισης του FTL. Στο παραπάνω σχήμα, δύο λογικές μονάδες σβησίματος αντιστοιχίζονται σε τρεις φυσικές μονάδες. Κάθε μονάδα σβησίματος περιέχει τέσσερις τομείς. Κάποιοι τομείς περιέχουν σελίδες αντιστοίχισης. Κάθε σελίδα αντιστοίχισης περιέχει τέσσερις αντιστοιχίσεις. Οι δείκτες, που εδώ αναπαρίστανται με γκρι τετράγωνα, αποθηκεύονται στη RAM. Οι πίνακες αντιστοίχισης εικονικών σε λογικές σελίδες (virtual-to-logical page maps), που φαίνονται πάνω δεξιά, δεν αποθηκεύονται σε συνεχόμενες θέσεις μνήμης, οπότε ένας πίνακας στη RAM αντιστοιχίζει τους τομείς τους. Ο εφεδρικός πίνακας (replacement map) περιέχει μόνο έναν τομέα. Δεν έχει κάθε τομέας του αρχικού πίνακα (primary map), εγγραφή στον εφεδρικό πίνακα. Στην αντιστοίχιση του εικονικού μπλοκ 5, χρησιμοποιείται η εγγραφή του εφεδρικού πίνακα επειδή δεν είναι ελεύθερη (όλα τα bits 1).



Σχήμα 2.5: Παράδειγμα αντιστοίχισης στο FTL.

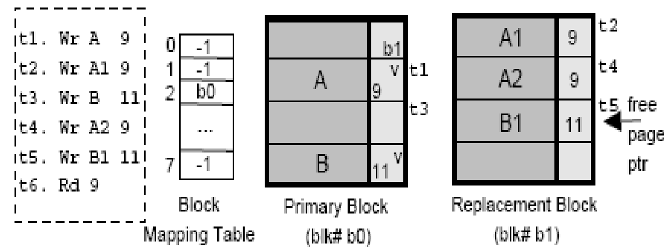
Το σχήμα 2.5 παρουσιάζει ένα παράδειγμα αντιστοίχισης μίας λογικής διεύθυνσης μπλοκ (LBA) σε μία φυσική διεύθυνση μπλοκ (PBA). Πιο συγκεκριμένα, η LBA ‘4’ αντιστοιχίζεται στην PBA ‘(0,5)’ μέσω του απευθείας πίνακα αντιστοίχισης/μετάφρασης που βρίσκεται στη RAM. Η LBA είναι μία διεύθυνση σελίδας που παρέχεται από το λειτουργικό σύστημα, ενώ κάθε PBA αποτελείται από δύο μέρη: (α) τον αριθμό του μπλοκ στον οποίο αντιστοιχίζεται η LBA και (β) τον αριθμό σελίδας του μπλοκ στον οποίο αντιστοιχίζεται η LBA. Στο συγκεκριμένο παράδειγμα, η λογική διεύθυνση μπλοκ ‘4’ αντιστοιχίζεται στη σελίδα ‘5’ του μπλοκ ‘0’ της μνήμης φλας.

2.3.3 NAND Flash Translation Layer (NFTL)

2.3.3.1 Τεχνικές αντιστοίχισης μπλοκ του NFTL

Ένα δημοφιλές στρώμα μετάφρασης βασισμένο σε μπλοκ (block based translation layer) είναι το **NAND Flash Translation Layer (NFTL)**. Στο NFTL ο τομέας (λογική διεύθυνση μπλοκ) αποτελείται από την εικονική διεύθυνση του μπλοκ (τα πιο σημαντικά bits) και από το αντιστάθμισμα σελίδας (page offset) (τα λιγότερο σημαντικά bits). Ένας πίνακας αντιστοίχισης μπλοκ αντιστοιχίζει την εικονική διεύθυνση του μπλοκ σε φυσικό μπλοκ. Το φυσικό αυτό μπλοκ είναι γνωστό και ως **βασικό/πρωταρχικό μπλοκ (primary block)**. Η πρώτη εγγραφή σε έναν τομέα πραγματοποιείται πάντοτε σε έναν τομέα του βασικού μπλοκ. Μεταγενέστερες ανανεώσεις (updates) στον ίδιο τομέα πραγματοποιούνται σε άλλο φυσικό μπλοκ γνωστό ως **μπλοκ αντικατάστασης (replacement block)**. Επιπρόσθετα, μία ανανέωση σε οποιαδήποτε σελίδα του βασικού μπλοκ πραγματοποιείται σε μία νέα σελίδα του μπλοκ αντικατάστασης. Κάθε βασικό μπλοκ, συσχετίζεται με ένα μπλοκ αντικατάστασης και οι εγγραφές στο μπλοκ αντικατάστασης γίνονται ακολουθιακά ξεκινώντας από τη σελίδα 0. Στην περίπτωση των αναγνώσεων, το μπλοκ αντικατάστασης διαβάζεται από το τέλος προς την αρχή. Εάν μία σελίδα του επιθυμητού τομέα δεν βρεθεί στο μπλοκ αντικατάστασης, τότε αναζητείται στο βασικό μπλοκ. Κάθε σελίδα περιέχει και κάποια bytes ελέγχου (control bytes) γνωστά ως **Out of Band δεδομένα (OOB data)**. Τυπικά, το μέγεθος των OOB δεδομένων είναι συνήθως 16 bytes και χρησιμοποιούνται κυρίως για αποθήκευση κώδικα διόρθωσης λαθών (Error Correcting Code-ECC).

Το παράδειγμα που ακολουθεί παρουσιάζει το NFTL με μεγαλύτερη λεπτομέρεια. Ας θεωρήσουμε για παράδειγμα μία μνήμη φλας αποτελούμενη από οχτώ μπλοκ με τέσσερις σελίδες ανά μπλοκ. Έτσι, η διεύθυνση του μπλοκ αποτελείται από τρία bits ενώ το αντιστάθμισμα (offset) από δύο bits.



Σχήμα 2.6: NFTL.

Μία ακολουθία από αναγνώσεις και εγγραφές (t1,..,t6) παρουσιάζονται στο σχήμα 2.5. Η πράξη t1: Wr A 9 πραγματοποιεί εγγραφή του δεδομένου 'A' στον τομέα με αριθμό 9 (δυαδικά-01001). Συνεπώς, σύμφωνα με τα παραπάνω, η εικονική διεύθυνση μπλοκ είναι 010 (τα τρία πιο σημαντικά bits του τομέα) και το αντιστάθμισμα σελίδας (page offset) είναι 01. Με βάση τον πίνακα αντιστοίχισης μπλοκ (σχήμα 2.6 Block Mapping Table) και την εικονική διεύθυνση μπλοκ, εντοπίζεται ένα ελεύθερο φυσικό μπλοκ (με αριθμό μπλοκ b0) το οποίο ανάγεται ως βασικό μπλοκ (primary block) για το εικονικό μπλοκ 010. Στη συνέχεια, το δεδομένο 'A' εγγράφεται στη σελίδα 1 του μπλοκ b0. Ο αριθμός του τομέα (στη συγκεκριμένη περίπτωση το 9) γράφεται στην περιοχή των OOB δεδομένων η οποία σημειώνεται ως έγκυρη.

Πράξη t2: Wr A1 9. Από τη στιγμή που δεν μπορούμε να επανεγγράψουμε τη σελίδα 1 του μπλοκ b0, εντοπίζεται ένα νέο φυσικό μπλοκ (με αριθμό μπλοκ b1) το οποίο και ανάγεται ως μπλοκ αντικατάστασης (replacement block) για το εικονικό μπλοκ 010. Το δεδομένο 'A1' εγγράφεται στην πρώτη διαθέσιμη/ελεύθερη σελίδα του μπλοκ αντικατάστασης, δηλαδή στο συγκεκριμένο παράδειγμα στη σελίδα 0. Η πληροφορία για το μπλοκ αντικατάστασης, αποθηκεύεται σε μία προκαθορισμένη περιοχή στο βασικό μπλοκ και ονομάζεται περιοχή OOB δεδομένων 0 (OOB data area 0). Πράξη t3: Wr B 11. Η περιοχή δεδομένων OOB που σχετίζεται με το αντιστάθμισμα σελίδας (page offset) 11 διαβάζεται για να δούμε αν ήδη περιέχει κάποια έγκυρα δεδομένα (σε μία τέτοια περίπτωση η εγγραφή πραγματοποιείται στο μπλοκ αντικατάστασης). Στο συγκεκριμένο παράδειγμα η περιοχή αυτή περιέχει έγκυρα δεδομένα και μάλιστα το δεδομένο αυτό είναι το B.

Οι πράξεις t4, t5 είναι ανανεώσεις συνεπώς τα νέα δεδομένα 'A2' και 'B1' εγγράφονται αντίστοιχα σε σελίδες του μπλοκ αντικατάστασης b1. Η πράξη t6: Rd 9 είναι μία πράξη ανάγνωσης του τομέα με αριθμό 9. Ο πίνακας αντιστοίχισης μπλοκ, μας οδηγεί στο φυσικό μπλοκ b0. Διαβάζοντας την επικεφαλίδα του μπλοκ b0 οδηγούμαστε στο μπλοκ αντικατάστασης b1. Το πιο πρόσφατο (έγκυρο) αντίγραφο των δεδομένων βρίσκεται ψάχνοντας στο μπλοκ αντικατάστασης από το τέλος προς την αρχή, ξεκινώντας από τη θέση που υποδεικνύει ο δείκτης ελεύθερων σελίδων (free page ptr). Εάν η σελίδα του επιθυμητού τομέα δεν βρεθεί στο μπλοκ αντικατάστασης, τότε βρίσκεται σίγουρα στο βασικό μπλοκ.

Με το πέρασμα του χρόνου, το μπλοκ αντικατάστασης μπορεί να γεμίσει. Σε αυτή την περίπτωση, επιλέγεται ένα νέο φυσικό μπλοκ και τα έγκυρα δεδομένα του βασικού μπλοκ και του μπλοκ αντικατάστασης συγχωνεύονται στο νέο φυσικό μπλοκ. Στη συνέχεια, το βασικό μπλοκ και το μπλοκ αντικατάστασης σβήνονται. Το νέο μπλοκ ανάγεται σε βασικό μπλοκ και ο πίνακας αντιστοίχισης μπλοκ τροποποιείται έτσι ώστε να περιέχει την παραπάνω αλλαγή. Η παραπάνω τεχνική της συγχώνευσης ενός φυσικού μπλοκ και του αντίστοιχου μπλοκ αντικατάστασης σε ένα νέο μπλοκ ονομάζεται **τεχνική σύμπτυξης (folding)**. Η χρήση του NFTL μπορεί να οδηγήσει σε μία κατάσταση όπου δεν υπάρχουν καθόλου ελεύθερα μπλοκ. Σε τέτοιες περιπτώσεις χρησιμοποιούνται **μηχανισμοί συλλογής απορριμμάτων (garbage collection mechanisms)** για την δημιουργία ελεύθερων μπλοκ.

Συγκεντρωτικά, θα μπορούσαμε να πούμε ότι το NFTL παρέχει τη δυνατότητα γρήγορων εγγραφών, με βασικό μειονέκτημα του μεγάλους χρόνους αναγνώσεων, οι οποίοι στη χειρότερη περίπτωση διαρκούν όσο και το ψάξιμο όλου του μπλοκ αντικατάστασης. Στην παρακάτω ενότητα, παρουσιάζονται δύο τεχνικές που έχουν ως σκοπό να βελτιώσουν τις επιδόσεις του NFTL τόσο στις αναγνώσεις δεδομένων όσο και στις εγγραφές.

2.3.3.2 Τεχνικές βελτίωσης της απόδοσης του NFTL

Σε αυτή την ενότητα θα παρουσιάσουμε δύο τεχνικές που επεκτείνουν το NFTL και βελτιώνουν τις επιδόσεις των βασικών λειτουργιών ανάγνωσης και εγγραφής.

2.3.3.2.1 Πίνακας αναζήτησης (Lookup Table)

Η πρώτη τεχνική περιλαμβάνει δύο δομές δεδομένων που αποθηκεύονται στη μνήμη RAM. Τον **πίνακα rep-block (rep-block table)** και τον **πίνακα κατάστασης σελίδας (page-status bitmap)**. Ο πίνακας rep-block επιταχύνει τη διαδικασία ευρέσεως του μπλοκ αντικατάστασης που σχετίζεται με ένα δοθέν βασικό μπλοκ. Ο πίνακας κατάστασης σελίδας επιταχύνει τη διαδικασία ελέγχου του κατά πόσο μία σελίδα περιέχει έγκυρα δεδομένα.

Ο πίνακας rep-block παρέχει γρήγορη πρόσβαση σε πληροφορίες του μπλοκ αντικατάστασης. Δεικτοδοτείται από τις εικονικές διευθύνσεις των μπλοκ. Για ένα δοθέν εικονικό μπλοκ, μία εγγραφή του πίνακα rep-block περιέχει τη φυσική διεύθυνση του μπλοκ αντικατάστασης. Με τον πίνακα rep-block, αποφεύγεται ο φόρτος ανάγνωσης δεδομένων OOB μιας και οι αναζητήσεις γίνονται πια σε επίπεδο RAM (όπως προαναφέραμε ο πίνακας rep-block βρίσκεται στη RAM) με αποτέλεσμα την επιτάχυνσή τους. Ο πίνακας rep-block βελτιώνει την απόδοση του NFTL, αποφεύγοντας τις αναγνώσεις OOB δεδομένων σε επίπεδο μνήμης φλας στις παρακάτω περιπτώσεις:

- Κάθε λειτουργία ανάγνωσης σελίδας, προκαλεί την ανάγνωση δεδομένων OOB με σκοπό να εξεταστεί η ύπαρξη μπλοκ αντικατάστασης.

- Η επανεγγραφή σε μία σελίδα απαιτεί τη γνώση της φυσικής διεύθυνσης του μπλοκ αντικατάστασης.

- Επιταχύνεται η διαδικασία συλλογής απορριμμάτων μιάς και ο πίνακας rep-block παρέχει τη φυσική διεύθυνση του μπλοκ αντικατάστασης για κάθε βασικό μπλοκ που πρέπει να συμπτυχθεί (folded).

Ο πίνακας κατάστασης σελίδας (page-status bitmap), υποδεικνύει την κατάσταση κάθε σελίδας και δεικτοδοτείται από το αντιστάθμισμα κάθε σελίδας (page offset). Για ένα δοθέν αντιστάθμισμα σελίδας, η τιμή 1 υποδηλώνει ότι η αντίστοιχη σελίδα έχει υποστεί εγγραφή τουλάχιστον μία φορά, ενώ η τιμή 0 υποδηλώνει ότι η σελίδα δεν έχει γραφτεί καμία φορά. Αξίζει να αναφέρουμε ότι, η παραπάνω πληροφορία του πίνακα κατάστασης σελίδας μπορεί να οδηγήσει σε αποφυγή της ανάγνωσης δεδομένων OOB μιας και:

- Κάθε λειτουργία εγγραφής ελέγχει τα δεδομένα OOB για να αποφασιστεί το κατά πόσο η πράξη θα πραγματοποιηθεί στο βασικό μπλοκ ή στο μπλοκ αντικατάστασης.

- Κάθε λειτουργία ανάγνωσης ελέγχει τα δεδομένα OOB για να εξετάσει την πιθανότητας μίας παράνομης πράξης ανάγνωσης.

Συμπερασματικά λοιπόν θα λέγαμε ότι ο πίνακας κατάστασης σελίδας επιταχύνει τόσο τη λειτουργία ανάγνωσης όσο και τη λειτουργία εγγραφής δεδομένων.

Οι απαιτήσεις σε μνήμη του πίνακα rep-block είναι όμοιες με αυτές του πίνακα αντιστοίχισης μπλοκ. Οι αντίστοιχες απαιτήσεις μνήμης του πίνακα κατάστασης σελίδας είναι οι ελάχιστες δυνατές – 1 bit ανά σελίδα. Έτσι, σε μία μνήμη φλας μεγέθους S, με μέγεθος μπλοκ B και μέγεθος σελίδας P, ο πίνακας rep-block έχει (S/B) εγγραφές/θέσεις, ενώ ο πίνακας κατάστασης σελίδας απαιτεί (S/P) bits. Για παράδειγμα, μία μνήμη φλας μεγέθους 1 GB με μέγεθος σελίδας 512 bytes, με μέγεθος μπλοκ 8 KB και 4 bytes ανά οντότητα, απαιτεί $((2^{30}/2^{13})\times 4)$=512 KB μνήμης και ο πίνακας κατάστασης σελίδας απαιτεί $(2^{30}/(2^{9}\times 8))$=256 KB μνήμης.

2.3.3.2.2 Κρυφή μνήμη σελίδας (Page Cache)

Η κρυφή μνήμης σελίδας είναι μία ειδικά διαμορφωμένη κρυφή μνήμη στην οποία αποθηκεύονται οι φυσικές διευθύνσεις των μπλοκ και των σελίδων, των πιο πρόσφατα γραμμένων τομέων. Ο εντοπισμός μίας σελίδας ή ενός μπλοκ γίνεται απευθείας χωρίς να χρειάζεται η ανάγνωση δεδομένων OOB, η οποία στη χειρότερη περίπτωση θα μπορούσε να οδηγήσει σε αναγνώσεις δεδομένων OOB ίσες με τον αριθμό σελίδων ανά μπλοκ (κατά τη διάρκεια μίας πράξης ανάγνωσης). Παρόμοια με τον πίνακα αναζήτησης (lookup table), οι εγγραφές/καταχωρίσεις της κρυφής μνήμης σελίδας είναι αντίγραφα της πληροφορίας που αποθηκεύεται στην περιοχή δεδομένων OOB. Έτσι, δεν υπάρχει ανάγκη να μεταφέρουμε αυτή τη πληροφορία στη μνήμη φλας. Η κρυφή μνήμη σελίδας μπορεί να βελτιώσει τους χρόνους αντιστοίχισης από τομέα σε μπλοκ/σελίδα, στις παρακάτω περιπτώσεις:

- Κατά τη διάρκεια μίας πράξης ανάγνωσης τομέα.

- Κατά τη διάρκεια αναγνώσεων που προκαλούνται από τη λειτουργία της σύμπτυξης μπλοκ (folding).

Ο πίνακας 2.1 που ακολουθεί παρουσιάζει συνοπτικά το βαθμό στον οποίο μπορούν να βελτιώσουν την επίδοση βασικών λειτουργιών όπως, ανάγνωση, εγγραφή, επανεγγραφή, σύμπτυξη και συλλογή απορριμμάτων, οι δομές δεδομένων που παρουσιάστηκαν σε αυτή την ενότητα.

Technique	Read	Write	Rewrite	Fold	GC
rep-block	●	○	●	○	●
page bitmap	●	◗	●	○	○
page cache	◗	○	○	◗	○

● Always ○ Never ◗ May

Πίνακας 2.1: Δυνατότητες βελτίωσης βασικών λειτουργιών του NFTL.

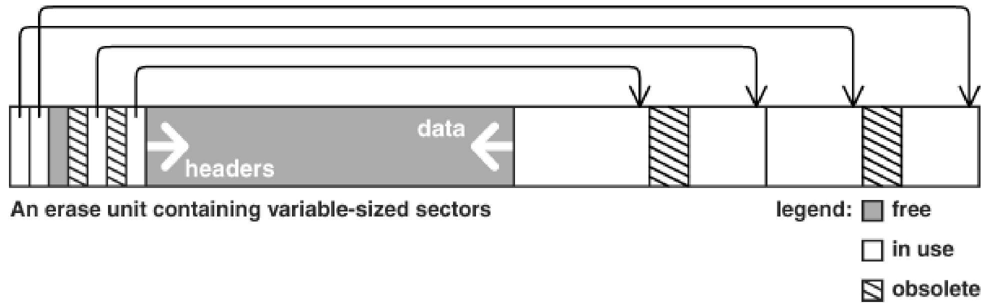
2.3.4 Αντιστοίχιση μπλοκ με μεταβλητό μέγεθος

Μία άλλη κατηγορία αντιστοίχισης μπλοκ είναι αυτή της αντιστοίχισης λογικών μπλοκ μεταβλητού μεγέθους στη μνήμη φλας. Οι Wells, Hasbun και Robinson εισήγαγαν πρώτοι την παραπάνω τεχνική η οποία στη συνέχεια υιοθετήθηκε με κάποιες τροποποιήσεις και από το σύστημα αρχείων για μνήμη φλας της Microsoft (Microsoft Flash File System). Το κίνητρο για τους Wells, Hasbun και Robinson ήταν η συμπιεσμένη αποθήκευση σε πρότυπους τομείς δίσκων (standard disk sectors). Για παράδειγμα, εάν τα τελευταία 200 bytes ενός τομέα των 512 bytes είναι όλα μηδέν, τότε τα μηδενικά μπορούν να αναπαρασταθούν «σιωπηρά» (implicitly) παρά «κατηγορηματικά» (explicitly) εξοικονομώντας έτσι αποθηκευτικό χώρο.

Η βασική ιδέα πίσω από την παραπάνω τεχνική είναι η τοποθέτηση μπλοκ δεδομένων μεταβλητού μεγέθους στο ένα άκρο μίας μονάδας σβησίματος και επικεφαλίδων καθορισμένου μεγέθους (fixed-size headers) στο άλλο άκρο. Κάθε επικεφαλίδα περιλαμβάνει έναν δείκτη προς το μπλοκ δεδομένων μεταβλητού μεγέθους που αντιπροσωπεύει. Οι επικεφαλίδες καθορισμένου μεγέθους επιτρέπουν προσπέλαση δεδομένων σε σταθερό χρόνο (στην πρώτη λέξη των δεδομένων).

Οι Smith και Garvin δημιούργησαν ένα ανάλογο σύστημα το οποίο χωρίζει κάθε μονάδα σβησίματος σε μία επικεφαλίδα (header), έναν πίνακα ανάθεσης (allocation map) και αρκετούς τομείς καθορισμένου μεγέθους (fixed-size sectors). Το σύστημα αναθέτει χώρο σε μπλοκ τα οποία αποτελούνται από έναν ή και περισσότερους συνεχόμενους τομείς. Τέτοια μπλοκ αποκαλούνται **επεκτάσεις (extents)**. Για κάθε επέκταση (μπλοκ) στην οποία έχει ανατεθεί χώρος, υπάρχει μία καταχώριση στον πίνακα ανάθεσης. Η καταχώριση καθορίζει την τοποθεσία και το μέγεθος της επέκτασης καθώς και τον εικονικό αριθμό του μπλοκ του πρώτου τομέα της επέκτασης (οι άλλοι τομείς της επέκτασης αποθηκεύουν συνεχόμενα εικονικά μπλοκ). Όταν ένα εικονικό μπλοκ μίας επέκτασης ανανεώνεται (updated), η επέκταση «σπάει» σε δύο ή τρεις νέες επεκτάσεις μία εκ των οποίων περιέχει το απαρχαιωμένο (obsolete) μπλοκ. Η αρχική καταχώριση της επέκτασης στον πίνακα ανάθεσης

σημειώνεται ως μη έγκυρη (invalid) και μία ή δύο νέες καταχωρίσεις προστίθενται στο τέλος του πίνακα ανάθεσης.



Σχήμα 2.7: Απεικόνιση μπλοκ με τομείς μεταβλητού μεγέθους. Επικεφαλίδες καθορισμένου μεγέθους προστίθενται στο ένα άκρο της μονάδας σβησίματος ενώ τομείς μεταβλητού μεγέθους προστίθενται στο άλλο άκρο. Το σχήμα δείχνει μία μονάδα σβησίματος με τέσσερις έγκυρους τομείς, δύο απαρχαιωμένους (obsolete) καθώς και κάποιο ελεύθερο χώρο (συμπεριλαμβανομένης μίας ελεύθερης επικεφαλίδας, της τρίτης κατά σειρά στο σχήμα).

Βιβλιογραφία κεφαλαίου

[1] Eran Gal and Sivan Toledo. Algorithms and Data Structures for Flash Memories. (χρήση στις σελίδες 17-20, 21-22, 27-28)

[2] Siddhart Choudhuri and Tony Givargis. Performance Improvement of Block Based NAND Flash Translation Layer. (χρήση στις σελίδες 23-27)

[3] Yuan-Hao Chang, Jen-Wei Hsieh and Tei-Wei Kuo. Endurance Enhancement of Flash-Memory Storage Systems: An Efficient Static Wear Levelling Design. (χρήση στις σελίδες 20, 23)

Κεφάλαιο 3

Συστήματα αρχείων ειδικά για μνήμη φλας

Περιεχόμενα

3.1 Εισαγωγή……………………………………………………………….. 30

3.2 Συστήματα αρχείων με δομή Log (Ημερολογίου)

(Log-Structured File Systems)……………………………………… 31

3.3 Journaling Flash File System (JFFS)……………………………… 34

3.3.1 JFFSv1………………………………………………………………… 34

3.3.1.1 Οργάνωση αποθήκευσης (Storage format)... 34

3.3.1.2 Λειτουργίες του JFFSv1.. 35

3.3.1.3 Συλλογή απορριμμάτων (Garbage Collection).................................... 35

3.3.2 JFFSv2………………………………………………………………… 36

3.3.2.1 Οργάνωση αποθήκευσης (Storage format)... 37

3.3.2.2 Εξισορρόπηση φθοράς (Wear Leveling).. 37

3.4 YAFFS: Yet Another Flash File System…………………………….38

3.4.1 Οργάνωση αποθήκευσης (Storage format)……………………………...39

3.4.2 YAFFSv1……………………………………………………………...39

3.4.3 YAFFSv2……………………………………………………………..40

3.4.4 Εξισορρόπηση φθοράς (Wear Leveling)………………………………..40

3.4.5 Σύγκριση YAFFS και JFFSv2…………………………………………..41

3.5 Trimble File System…………………………………………………..41

3.5.1 Οργάνωση αποθήκευσης (Storage format)……………………………..42

3.5.2 Ανάκτηση μονάδων σβησίματος (erase units reclamation)…………….. 42

3.6 Research-In-Motion File System..…………………………………... 42

3.6.1 Οργάνωση αποθήκευσης (Storage format)…………………………….. 43

Βιβλιογραφία κεφαλαίου.. 43

3.1 Εισαγωγή

Οι τεχνικές αντιστοίχισης μπλοκ που αναλύθηκαν στο κεφάλαιο 2, παρουσιάζουν τη συσκευή φλας σε ένα πιο υψηλό επίπεδο λογισμικού, σε συγκεκριμένα συστήματα αρχείων (όπως το FAT), σαν μία επανεγγράψιμη συσκευή μπλοκ. Ο οδηγός της συσκευής μπλοκ (block device driver) πραγματοποιεί λειτουργίες όπως αντιστοίχιση μπλοκ σε τομείς (block-to-sector mapping), ανάκτηση μονάδων σβησίματος (erase-unit reclamation), εξισορρόπηση φθοράς (wear leveling) καθώς και αποκατάσταση της συσκευής μπλοκ μετά από μία κατάσταση κατάρρευσης (crash). Μία άλλη προσέγγιση, είναι να εκθέσουμε τα hardware χαρακτηριστικά της μνήμης φλας στο επίπεδο του συστήματος αρχείων και να το αφήσουμε να διαχειριστεί τις μονάδες σβησίματος καθώς και τη φθορά των μπλοκ. Το βασικό κίνητρο πίσω από αυτή την ιδέα είναι ότι μία ολική λύση (end-to-end solution) θα μπορούσε να είναι πιο αποδοτική από το να τοποθετούμε ένα σύστημα αρχείων, σχεδιασμένο για μαγνητικούς σκληρούς δίσκους, στην κορυφή μίας συσκευής που είναι σχεδιασμένη να λειτουργεί όπως οι δίσκοι, χρησιμοποιώντας όμως μνήμη φλας.

Οι μηχανισμοί αντιστοίχισης μπλοκ προσφέρουν αρκετά πλεονεκτήματα στα **συστήματα αρχείων ειδικά για μνήμη φλας (flash-specific file systems)**. Πρώτον, η προσέγγιση της αντιστοίχισης μπλοκ επιτρέπει στους κατασκευαστές να αξιοποιούν ήδη υπάρχοντα συστήματα αρχείων αποφεύγοντας με αυτόν τον τρόπο κατασκευαστικές δαπάνες καθώς και κόστη δοκιμών και ελέγχων. Δεύτερον, οι **αφαιρούμενες συσκευές φλας (removable flash devices)**, όπως για παράδειγμα η CompactFlash και η SmartMedia, πρέπει να χρησιμοποιούν **οργάνωση/διαμόρφωση αποθήκευσης (storage format)** η οποία να είναι συμβατή με όλες τις πλατφόρμες στις οποίες πιθανόν οι χρήστες θα τις χρησιμοποιήσουν. Οι πλατφόρμες αυτές περιλαμβάνουν λειτουργικά συστήματα όπως Windows, Mac και Linux/Unix καθώς επίσης και συσκευές παλάμης (hand-held devices) όπως ψηφιακές κάμερες, κινητά τηλέφωνα, PDAs και συσκευές αναπαραγωγής ήχου (music players). Το μόνο επανεγγράψιμο σύστημα αρχείων που υποστηρίζεται από τα περισσότερα λειτουργικά συστήματα είναι το FAT. Για αυτό το λόγο, οι αφαιρούμενες συσκευές το χρησιμοποιούν τυπικά. Εάν η αφαιρούμενη συσκευή χρησιμοποιεί τη μνήμη φλας απευθείας, τότε το σύστημα host πρέπει να κατανοεί τις επικεφαλίδες οι οποίες περιγράφουν το περιεχόμενο των μονάδων σβησίματος και των τομέων. Η επικράτηση του FTL (Flash Translation Layer), επιτρέπει σε αυτές τις επικεφαλίδες να υφίστανται επεξεργασία σε πολλαπλές πλατφόρμες. Η προσέγγιση αυτή χρησιμοποιείται και από το πρότυπο της SmartMedia. Μία άλλη προσέγγιση που καθιερώθηκε από την SanDisk, είναι να «κρύψουμε» τη συσκευή μνήμης φλας πίσω από μία διεπαφή δίσκου (disk interface), η οποία υλοποιείται σε επίπεδο υλικού σαν μέρος της αφαιρούμενης συσκευής, απαλλάσσοντας το σύστημα host από την κατανόηση των επικεφαλίδων.

Όταν μία αφαιρούμενη συσκευή χρησιμοποιεί μία ήδη υπάρχουσα δομή συστήματος αρχείων, όπως για παράδειγμα το FAT, ο συνδυασμός του συστήματος αρχείων με μηχανισμούς αντιστοίχισης μπλοκ, προσφέρει αρκετά πλεονεκτήματα. Ας πάρουμε για παράδειγμα την διαγραφή ενός αρχείου. Εάν το σύστημα χρησιμοποιεί ένα προαναφερθέν σύστημα αρχείων σε συνδυασμό με ένα μηχανισμό αντιστοίχισης μπλοκ, οι τομείς που περιέχουν τα διαγραμμένα αρχεία δεδομένων, είτε θα σημειωθούν ως ελεύθεροι σε έναν πίνακα-bit (bitmap table), είτε θα σημειωθούν ως απαρχαιωμένοι (obsolete) αλλά δεν θα πανωγραφτούν (overwritten). Πράγματι, όταν

το σύστημα αρχείων αποθηκεύεται σε ένα μαγνητικό δίσκο, δεν υπάρχει λόγος να μετακινούμε τον δείκτη ανάγνωσης-εγγραφής (read-write head) στους τομείς που προέκυψαν ελεύθεροι εξαιτίας της παραπάνω διαγραφής. Αλλά αν το σύστημα αρχείων αποθηκεύεται σε μία συσκευή φλας, τα διαγραμμένα μπλοκ δεδομένων που δε σημειώνονται ως απαρχαιωμένα, αντιγράφονται από τη μία μονάδα στην άλλη, κάθε φορά που ανακτάται η μονάδα που τα περιέχει. Συνδυάζοντας όμως τις τεχνικές αντιστοίχισης μπλοκ με το λογισμικό του συστήματος αρχείων, το σύστημα μπορεί να σημειώσει τους τομείς των διαγραμμένων αρχείων ως απαρχαιωμένους, γεγονός που τους προστατεύει από το να αντιγράφονται. Η προσέγγιση αυτή χρησιμοποιείται από το FLite της M-Systems, το οποίο είναι μία παραλλαγή του FAT συστήματος αρχείων.

Όταν όμως η συσκευή μνήμης φλας δεν είναι αφαιρούμενη, τότε η χρησιμοποίηση ενός **συστήματος αρχείων ειδικό για μνήμη φλας (flash-specific file system)** είναι η καλύτερη επιλογή. Τα περισσότερα συστήματα αρχείων ειδικά για μνήμη φλας έχουν ως βασική τους αρχή το **σύστημα αρχείων με δομή log (ημερολογίου) (log-structured file system)**. Ως εκ τούτου, στην επόμενη ενότητα ακολουθεί η περιγραφή του συστήματος αρχείων με δομή log και στη συνέχεια παρουσιάζονται ορισμένα βασικά συστήματα αρχείων ειδικά για μνήμη φλας.

3.2 Συστήματα αρχείων με δομή Log (Ημερολογίου) (Log-Structured File Systems)

Στα παραδοσιακά συστήματα αρχείων, οι τροποποιήσεις των δεδομένων γίνονται στις ίδιες ακριβώς θέσεις μνήμης (in place modifications). Όταν ένα μπλοκ δεδομένων, που περιέχει είτε ένα μέρος ενός αρχείου είτε μεταδεδομένα του συστήματος αρχείων πρέπει να υποστεί τροποποίηση, τότε τα νέα δεδομένα πανωγράφουν (overwrite) τα παλαιά στις ίδιες ακριβώς φυσικές τοποθεσίες στο δίσκο. Η διαδικασία αυτή διευκολύνει τη διεργασία ευρέσεως δεδομένων, μιας και η δομή δεδομένων που χρησιμοποιούνταν για την εύρεση των παλαιών δεδομένων χρησιμοποιείται και για την εύρεση των τροποποιημένων δεδομένων. Για παράδειγμα, εάν το τροποποιημένο μπλοκ περιείχε ένα μέρος ενός αρχείου, ο δείκτης που έδειχνε στα παλαιά δεδομένα συνεχίζει να είναι έγκυρος. Με άλλα λόγια, στα παραδοσιακά συστήματα αρχείων, οι δείκτες παραμένουν έγκυροι και μετά τις ανανεώσεις δεδομένων μιάς και τα δεδομένα (συμπεριλαμβανομένων τόσο μεταδεδομένων όσο και δεδομένων χρηστών) είναι ευμετάβλητα (mutable).

Οι εντός θέσης τροποποιήσεις δεδομένων (in place modifications) προκαλούν δύο βασικά προβλήματα. Το πρώτο είδος προβλημάτων περιέχει την συνεχή μετακίνηση του δείκτη ανάγνωσης-εγγραφής και την περιστροφή του δίσκου (disk rotation). Οι τροποποιήσεις δεδομένων αναγκάζουν το σύστημα να γράψει τα νέα δεδομένα σε συγκεκριμένες θέσεις του δίσκου με αποτέλεσμα αυτές οι εγγραφές να υποφέρουν από μεγάλους χρόνους αναζήτησης και από καθυστερήσεις περιστροφών δίσκου. Η επίδραση των παραπάνω καθυστερήσεων είναι ιδιαίτερα σημαντική κυρίως όταν η διαδικασία της εγγραφής δεν πρέπει να καθυστερεί, για παράδειγμα κατά το κλείσιμο ενός αρχείου. Το δεύτερο είδος προβλήματος που προκαλείται από τις εντός θέσης τροποποιήσεις δεδομένων, είναι η κατάσταση στην οποία θα βρεθούν τα μεταδεδομένα μετά από μία κατάρρευση (crash). Ακόμα και αν οι εγγραφές είναι ατομικές, κάτι τέτοιο δε συμβαίνει με τις λειτουργίες υψηλού επιπέδου που

αποτελούνται από πολλαπλές εγγραφές, όπως για παράδειγμα η δημιουργία ή η διαγραφή ενός αρχείου. Ως αποτέλεσμα των παραπάνω, οι δομές δεδομένων του συστήματος αρχείων μπορεί να βρεθούν σε μία ασυνεπή κατάσταση μετά από μία κατάρρευση. Υπάρχουν όμως αρκετές τεχνικές που αντιμετωπίζουν αυτή την ανεπιθύμητη κατάσταση.

Στα συστήματα αρχείων «άτακτης» εγγραφής (lazy-write file systems), μία διαδικασία τακτοποίησης (fixing-up process) διορθώνει την ασυνέπεια που προκαλείται από μία κατάσταση κατάρρευσης. Η διαδικασία τακτοποίησης καθυστερεί την ανάκαμψη του υπολογιστικού συστήματος από την κατάσταση κατάρρευσης. Στα συστήματα αρχείων «προσεχτικής» εγγραφής (careful-write file systems), οι αλλαγές στα μεταδεδομένα πραγματοποιούνται αυτομάτως σύμφωνα με μία αυστηρή σειρά, έτσι ώστε αμέσως μετά από μία κατάρρευση, οι δομές δεδομένων να είναι σε συνεπή κατάσταση εκτός ίσως από την απώλεια κάποιων μπλοκ δίσκου τα οποία ανακτώνται αργότερα από έναν συλλέκτη απορριμμάτων. Η προσέγγιση αυτή μειώνει το χρόνο ανάκαμψης του συστήματος αλλά ενισχύει την επίδραση των καθυστερήσεων δίσκου. Η διαδικασία των «μαλακών ανανεώσεων» (soft updates) είναι μία γενίκευση της «προσεχτικής» εγγραφής (careful-writing). Το σύστημα αρχείων μπορεί να αποθηκεύσει προσωρινά στην κρυφή μνήμη τις τροποποιήσεις των μπλοκ μεταδεδομένων και στη συνέχεια να τροποποιήσει τα μπλοκ, σύμφωνα με μία διάταξη η οποία εγγυάται ότι οι μόνες ασυνέπειες που προκαλούνται μετά από μία κατάσταση κατάρρευσης είναι πιθανόν κάποιες απώλειες μπλοκ δίσκου (disk blocks). Η διαδικασία των «μαλακών ανανεώσεων» προσεγγίζει αρκετά τη φιλοσοφία λειτουργίας των **συστημάτων αρχείων με ημερολόγιο (journaling file systems)**.

Στα συστήματα αρχείων με ημερολόγιο, κάθε τροποποίηση μεταδεδομένων γράφεται πρώτα σε ένα ημερολόγιο (journal) (ή ένα log), προτού το μπλοκ τροποποιηθεί εντός θέσης (in place modification). Αμέσως μετά από μία κατάσταση κατάρρευσης (crash), μία διαδικασία τακτοποίησης (fixing-up process) εξετάζει το τέλος του log και είτε ολοκληρώνει είτε αναστέλλει τις λειτουργίες των μεταδεδομένων που ίσως είχαν διακοπεί. Στο σημείο αυτό αξίζει να αναφέρουμε ότι εκτός από τα Sun Solaris συστήματα, τα οποία χρησιμοποιούν την διαδικασία των «μαλακών» ανανεώσεων, σχεδόν όλα τα εμπορικά λειτουργικά συστήματα χρησιμοποιούν συστήματα αρχείων με ημερολόγιο όπως για παράδειγμα το NTFS (New Technology File System) των Windows και το JFS (Journaled File System) του Linux.

Τα **συστήματα αρχείων με δομή Log (ημερολογίου) (Log-structured File Systems)** επεκτείνουν την αρχή λειτουργίας των συστημάτων αρχείων με ημερολόγιο (journalig file systems). Πιο συγκεκριμένα, το ημερολόγιο (journal) είναι το σύστημα αρχείων. Ο δίσκος οργανώνεται σαν ένα log, έναν συνεχόμενο αποθηκευτικό χώρο ο οποίος έχει εκ παραδοχής απεριόριστη χωρητικότητα. Στην πραγματικότητα, το log αποτελείται από τμήματα συνεχόμενων περιοχών δίσκου καθορισμένου μεγέθους, τα οποία συνδέονται μεταξύ τους σε μία συνδεδεμένη λίστα (linked list). Οι εγγραφές τόσο των δεδομένων όσο και των μεταδεδομένων γίνονται πάντα στο τέλος του log με αποτέλεσμα να μην πανωγράφονται (overwritten) παλαιά δεδομένα. Το γεγονός αυτό προκαλεί δύο βασικά προβλήματα: πώς γίνεται ο εντοπισμός των πρόσφατα γραμμένων (newly written) δεδομένων και πως γίνεται η ανάκτηση του χώρου που καταλαμβάνεται από απαρχαιωμένα (obsolete) δεδομένα.

Για τον εντοπισμό των πρόσφατα γραμμένων δεδομένων, απαιτείται τροποποίηση των δεικτών που «δείχνουν» προς αυτά τα δεδομένα, γεγονός που με τη σειρά του οδηγεί στην ανάγκη εγγραφής των μπλοκ που περιέχουν αυτούς τους δείκτες στο log. Για να περιοριστεί μία πιθανή «χιονοστιβάδα» εγγραφών, τα αρχεία χαρακτηρίζονται από έναν **λογικό αριθμό inode**. Οι λογικοί αριθμοί inode, αντιστοιχίζονται σε φυσικές θέσεις στο log μέσω ενός πίνακα που βρίσκεται στη μνήμη RAM και περιοδικά αποθηκεύεται στο log. Αμέσως μετά από μία κατάσταση κατάρρευσης, ο πίνακας μπορεί να ανακατασκευαστεί βρίσκοντας το πιο πρόσφατο αντίγραφο του πίνακα στο log. Στη συνέχεια, σαρώνεται το log από τη θέση που βρέθηκε ο πίνακας μέχρι το τέλος, για την εύρεση αρχείων των οποίων η θέση άλλαξε αμέσως μετά την αποθήκευση αυτού του αντιγράφου του πίνακα στο log.

Για την ανάκτηση αποθηκευτικού χώρου, μία **διαδικασία εκκαθάρισης (cleaner process)** αναλαμβάνει την εύρεση τμημάτων (segments) τα οποία περιέχουν μεγάλες ποσότητες απαρχαιωμένων δεδομένων. Στη συνέχεια, αντιγράφει τα έγκυρα δεδομένα αυτών των τμημάτων στο τέλος του log, διαγράφει εξολοκλήρου αυτά τα τμήματα και προσθέτει τον ελεύθερο χώρο που προκύπτει στο τέλος του log.

Το βασικό πλεονέκτημα των συστημάτων αρχείων με δομή log, είναι το ότι προσφέρουν καλές επιδόσεις εγγραφών, μιας και όλες οι εγγραφές δεδομένων πραγματοποιούνται πάντα στο τέλος του log με αποτέλεσμα να αποφεύγονται καθυστερήσεις αναζητήσεων και περιστροφών δίσκου (seek and rotational delays). Από την άλλη πλευρά, η λειτουργία ανάγνωσης μπορεί να είναι αργή μιάς και τα μπλοκ ενός αρχείου μπορεί να είναι διασκορπισμένα στο δίσκο, εάν βέβαια κάθε μπλοκ τροποποιήθηκε σε διαφορετική χρονική στιγμή. Επιπρόσθετα, η διαδικασία εκκαθάρισης μπορεί να επιβραδύνει επιπλέον το σύστημα εφόσον διαγράφει τμήματα του δίσκου (disk segments) που περιέχουν μεγάλες ποσότητες έγκυρων δεδομένων. Σαν αποτέλεσμα του παραπάνω μειονεκτήματος, τα συστήματα αρχείων με δομή log δεν χρησιμοποιούνται ευρέως στους σκληρούς δίσκους.

Στις συσκευές μνήμης φλας, από την άλλη πλευρά, τα συστήματα αρχείων με δομή log θεωρούνται τα πλέον κατάλληλα. Είναι γνωστό ότι στη μνήμη φλας τα παλαιά δεδομένα δεν μπορούν να πανωγραφτούν. Οι τροποποιήσεις των δεδομένων πρέπει να γραφτούν σε διαφορετικές θέσεις από αυτές των αρχικών αντιγράφων. Επιπρόσθετα, η χρήση συστήματος αρχείων με δομή log σε συσκευές μνήμης φλας, δεν επηρεάζει τις επιδόσεις ανάγνωσης, μιας και η μνήμη φλας επιτρέπει ομοιόμορφους χρόνους προσπέλασης. Ως αποτέλεσμα των παραπάνω, τα περισσότερα συστήματα αρχείων ειδικά για μνήμη φλας (flash-specific file systems) χρησιμοποιούν ως βασική τους αρχή, τη λειτουργία των συστημάτων αρχείων με δομή log. Πρώτοι οι Kawaguchi, Nishioka και Motoda, ανακάλυψαν την καταλληλότητα των συστημάτων αρχείων με δομή log για την μνήμη φλας. Όχι μόνο χρησιμοποίησαν την προσέγγιση της δομής log για τη δημιουργία ενός οδηγού συσκευής αντιστοίχισης μπλοκ (block-mapping device driver) αλλά επισήμαναν ότι αυτή η προσέγγιση είναι κατάλληλη για τη γενικότερη διαχείριση της μνήμης φλας.

Στις ενότητες που ακολουθούν, παρουσιάζονται τα βασικά χαρακτηριστικά ορισμένων σημαντικών συστημάτων αρχείων ειδικά για μνήμη φλας.

3.3 Journaling Flash File System (JFFS)

Το JFFS είναι ένα σύστημα αρχείων ειδικό για μνήμη φλας, το οποίο ανήκει στην κατηγορία των συστημάτων αρχείων με δομή log (log-structured file systems). Σχεδιάστηκε από την εταιρία Axis Communications AB στην Σουηδία και απευθύνεται αποκλειστικά σε συσκευές φλας ενσωματωμένων συστημάτων (flash devices in embedded systems). Το JFFS λαμβάνει υπόψιν του τους περιορισμούς που επιβάλλει η τεχνολογία φλας ενώ παράλληλα λειτουργεί απευθείας στα τσιπ της μνήμης φλας (flash chips). Αποτέλεσμα των παραπάνω είναι η αποφυγή της μη αποδοτικής χρήσης δύο συστημάτων αρχείων με ημερολόγιο, το ένα στην κορυφή του άλλου. Στις ενότητες που ακολουθούν, θα παρουσιαστούν οι δύο εκδόσεις του JFFS (JFFSv1 και JFFSv2). Στο σημείο αυτό, αξίζει να αναφέρουμε πως και οι δύο εκδόσεις του JFFS εστιάζουν κυρίως σε **NOR συσκευές μνήμης φλας** και ίσως να μην δουλεύουν αξιόπιστα σε NAND συσκευές μνήμης φλας.

3.3.1 JFFSv1

Οι σχεδιαστικοί στόχοι του JFFSv1 καθορίζονται από τα χαρακτηριστικά της τεχνολογίας φλας καθώς και από τις συσκευές στις οποίες πρόκειται να εφαρμοστεί. Σε αυτές τις συσκευές ανήκουν τα ενσωματωμένα συστήματα (embedded systems) καθώς και συσκευές μπαταρίας (battery-powered devices). Το JFFSv1 πρέπει να παρέχει αξιόπιστες λειτουργίες όποτε οι παραπάνω συσκευές δεν τερματίζονται (shut down) με τον προβλεπόμενο τρόπο.

3.3.1.1 Οργάνωση αποθήκευσης (Storage format)

Το JFFSv1 ανήκει αμιγώς στην κατηγορία συστημάτων αρχείων με δομή log (log-structured file systems). Οι κόμβοι (nodes) που περιέχουν δεδομένα και μεταδεδομένα αποθηκεύονται στη μνήμη φλας σειριακά. Υπάρχει μόνο ένας τύπος κόμβων στο log (ημερολόγιο). Ουσιαστικά πρόκειται για μία δομή δεδομένων γνωστή ως **δομή jffs_ανεπεξέργαστου_inode (struct jffs_raw_inode)**. Κάθε τέτοιος κόμβος συσχετίζεται με ένα μοναδικό inode. Ο κόμβος ξεκινάει πάντα με μία κεφαλίδα (header) η οποία περιλαμβάνει, τον αριθμό του inode στο οποίο ανήκει, όλα τα μεταδεδομένα του συστήματος αρχείων που σχετίζονται με αυτό το inode καθώς και μία ποσότητα δεδομένων.

Υπάρχει μία ολική διάταξη (total ordering) μεταξύ όλων των κόμβων που ανήκουν σε ένα inode, η οποία επιτυγχάνεται με την ανάθεση ενός **αριθμού έκδοσης (version number)** σε κάθε κόμβο. Κάθε κόμβος που γράφεται στο inode, διαθέτει αριθμό έκδοσης μεγαλύτερο από αυτούς των κόμβων που είναι ήδη αποθηκευμένοι στο inode. Ο αριθμός έκδοσης είναι ένας μη προσημασμένος (unsigned) αριθμός των 32-bit με αποτέλεσμα να μπορούν να αποθηκεύονται τέσσερα δισεκατομμύρια κόμβοι σε ένα inode, κατά τη διάρκεια ζωής του συστήματος αρχείων. Εξαιτίας όμως της περιορισμένης «ζωής» (limited lifetime) της μνήμης φλας, ο παραπάνω αριθμός είναι πρακτικά αδύνατο να συμβεί στην πράξη. Παρομοίως, ο αριθμός inode αποθηκεύεται σε ένα πεδίο των 32 bit και δεν επαναχρησιμοποιείται (reused).

Κάθε ανεπεξέργαστος κόμβος (raw inode) του JFFSv1 περιλαμβάνει, το όνομα του inode στο οποίο ανήκει καθώς και τον αριθμό inode του πατέρα inode (parent inode). Επιπρόσθετα, κάθε κόμβος μπορεί να περιέχει μία ποσότητα δεομένων, με αποτέλεσμα ο κόμβος να καταγράφει (record) το αντιστάθμισμα (offset) στο αρχείο, στο οποίο πρέπει να εμφανίζονται αυτά τα δεδομένα. Μεγάλα αρχεία, συσχετίζονται με πολλούς κόμβους, μιάς και υπάρχει περιορισμός στο μέγιστο μέγεθος των φυσικών κόμβων (physical nodes). Πιο συγκεκριμένα, διαφορετικές εμβέλειες (ranges) του μεγάλου αρχείου αποθηκεύονται σε διαφορετικούς κόμβους. Υπάρχουν δύο περιπτώσεις κατά τις οποίες ένας κόμβος χαρακτηρίζεται **απαρχαιωμένος (obsoleted)**. Στην πρώτη κατηγορία, ανήκουν οι κόμβοι που περιέχουν δεδομένα μίας εμβέλειας ενός αρχείου, η οποία εμβέλεια αποθηκεύεται και σε έναν μεταγενέστερο κόμβο (later node). Στην δεύτερη κατηγορία, ανήκουν οι κόμβοι που δεν περιέχουν καθόλου δεδομένα, ενώ τα μεταδεδομένα τα οποία περιέχουν έχουν ξεπεραστεί (outdated) από μεταγενέστερους κόμβους. Ο χώρος που καταλαμβάνεται από απαρχαιωμένους κόμβους ονομάζεται **«βρώμικος» χώρος (dirty space)**.

Η διαγραφή του inode πραγματοποιείται θέτοντας μία σημαία (flag) ως **διαγραμμένη (deleted)** στα μεταδεδομένα του inode. Όλοι οι μεταγενέστεροι κόμβοι (later nodes), σχετιζόμενοι με το διαγραμμένο inode, σημειώνονται (marked) με την ίδια σημαία (flag) και όταν κλείσει και το τελευταίο αρχείο του inode, όλοι οι κόμβοι του inode προκύπτουν απαρχαιωμένοι (obsoleted).

3.3.1.2 Λειτουργίες του JFFSv1

Κατά το χρόνο προσάρτησης, γίνεται σάρωση ολόκληρου του αποθηκευτικού μέσου (μνήμη φλας) και όλοι οι κόμβοι διαβάζονται και ερμηνεύονται (interpreted). Τα δεδομένα που αποθηκεύονται στους ανεπεξέργαστους κόμβους (raw nodes), παρέχουν επαρκείς πληροφορίες για την ανακατασκευή ολόκληρης της ιεραρχίας ευρετηρίου (entire directory hierarchy) καθώς και της αντιστοίχισης των inode σε φυσικές διευθύνσεις/τοποθεσίες στη μνήμη φλας.

Κάθε αναζήτηση ευρετηρίου, ικανοποιείται απευθείας από δομές δεδομένων που βρίσκονται στον πυρήνα (in-core data structures), ενώ οι αναγνώσεις αρχείων πραγματοποιούνται διαβάζοντας απευθείας από τις κατάλληλες θέσεις της μνήμης φλας. Αλλαγές σε επίπεδο μεταδεδομένων, όπως αλλαγή της ιδιοκτησίας ενός αρχείου, πραγματοποιούνται γράφοντας ένα νέο κόμβο στο τέλος του log (ημερολογίου) ο οποίος περιέχει τα καινούργια μεταδεδομένα. Τέλος, οι εγγραφές σε επίπεδο αρχείων, πραγματοποιούνται με παρόμοιο τρόπο, με τη διαφορά ότι ο κόμβος που γράφεται περιέχει δεδομένα που σχετίζονται με το συγκεκριμένο αρχείο.

3.3.1.3 Συλλογή απορριμμάτων (Garbage collection)

Μέχρι στιγμής, οι αρχές όλων των λειτουργιών που παρουσιάστηκαν, είναι πάρα πολύ απλές. Ο κώδικας του JFFS δημιουργεί νέες δομές jffs_ανεπεξέργαστου_inode (jffs_raw_inode structures) στη μνήμη φλας, για να συμπεριλάβει οποιαδήποτε αλλαγή πραγματοποιείται στο σύστημα αρχείων, μέχρι το σημείο που δεν υπάρχει άλλος ελεύθερος αποθηκευτικός χώρος. Από το σημείο αυτό και μετά, το σύστημα

πρέπει να ανακτήσει το «βρώμικο» χώρο (dirty space) ο οποίος καταλαμβάνεται από απαρχαιωμένους κόμβους.

Ο παλαιότερος κόμβος στο log, είναι γνωστός ως **κεφαλή (head)** ενώ οι νέοι κόμβοι που δημιουργούνται προστίθενται στην **ουρά (tail)** του log. Σε ένα καθαρό σύστημα αρχείων, στο οποίο δεν έχει ακόμη επέμβει ο μηχανισμός συλλογής απορριμμάτων, η κεφαλή του log βρίσκεται στην αρχή της μνήμης φλας. Καθώς όμως η ουρά πλησιάζει το τέλος της μνήμης φλας, ο μηχανισμός συλλογής απορριμμάτων ενεργοποιείται για να ανακτήσει αποθηκευτικό χώρο.

Ο μηχανισμός συλλογής απορριμμάτων εκκινείτε, είτε στα πλαίσια μίας διεργασίας πυρήνα (kernel process) η οποία προσπαθεί να ανακτήσει εκ των προτέρων χώρο, είτε στα πλαίσια μίας διεργασίας χρήστη (user process) η οποία δεν βρίσκει κατάλληλο ελεύθερο χώρο στη μνήμη φλας για να πραγματοποιήσει μία επιθυμητή λειτουργία εγγραφής. Και στις δύο περιπτώσεις, ο μηχανισμός συλλογής απορριμμάτων θα συνεχίσει μόνο εάν υπάρχει «βρώμικος» χώρος (dirty space) που μπορεί να ανακτηθεί. Εάν όμως δεν υπάρχει αρκετός «βρώμικος» χώρος που να εγγυάται ότι ο μηχανισμός συλλογής απορριμμάτων θα βελτιώσει την κατάσταση, τότε η διεργασία πυρήνα (kernel process) θα κοιμηθεί (sleep) και οι εγγραφές θα αποτύχουν.

Ο στόχος του κώδικα συλλογής απορριμμάτων είναι να σβήσει το πρώτο μπλοκ της μνήμης φλας στο log. Σε κάθε πέρασμα, εξετάζεται ο κόμβος στην κεφαλή (head) του log. Εάν ο κόμβος είναι απαρχαιωμένος, τότε παρακάμπτεται (skipped) και ο αμέσως επόμενος κόμβος καθίσταται ως κεφαλή (head). Εάν όμως ο κόμβος είναι ακόμα έγκυρος (valid), πρέπει να μετατραπεί σε απαρχαιωμένο, μία διαδικασία που αναλαμβάνει ο κώδικας συλλογής απορριμμάτων γράφοντας έναν νέο κόμβο δεδομένων ή μεταδεδομένων στην ουρά (tail) του log.

Ο νέος κόμβος θα περιέχει τα έγκυρα δεδομένα, τουλάχιστον της εμβέλειας (range) που καλύπτεται από τον αρχικό κόμβο (original node). Εάν υπάρχει ικανοποιητική ποσότητα ελεύθερου χώρου, ο κώδικας συλλογής απορριμμάτων μπορεί να γράψει έναν μεγαλύτερο κόμβο από αυτόν που κατέστησε απαρχαιωμένο, με σκοπό να βελτιώσει την απόδοση αποθήκευσης, συγχωνεύοντας πολλούς μικρούς κόμβους σε λιγότερους αλλά μεγαλύτερους.

Εάν ο κόμβος που μετατράπηκε σε απαρχαιωμένο, ήταν από πριν μερικώς απαρχαιωμένος από μεταγενέστερους κόμβους (later nodes) οι οποίοι καλύπτουν μόνο ένα μέρος της ίδιας εμβέλειας δεδομένων, τότε κάποια δεδομένα που γράφονται στον νέο κόμβο, προφανώς θα διαφέρουν από τα δεδομένα που υπήρχαν στον αρχικό κόμβο (original node). Κατ'αυτόν τον τρόπο, ο κώδικας συλλογής απορριμμάτων επεξεργάζεται την κεφαλή του log (ημερολογίου) μέχρις ότου μία μονάδα σβησίματος (erase unit) να γίνει απαρχαιωμένη (obsolete), να σβηστεί και να γίνει διαθέσιμη για χρήση από την ουρά (tail) του log.

3.3.2 JFFSv2

Το JFFSv2 είναι μία βελτιωμένη έκδοση του JFFSv1, η οποία δημιουργήθηκε από τον David Woodhouse της εταιρίας Red Hat.

3.3.2.1 Οργάνωση αποθήκευσης (Storage format)

Στο JFFSv2, τα αρχεία χαρακτηρίζονται από έναν **αριθμό inode**. Οι αριθμοί inode δεν επαναχρησιμοποιούνται (reused) και κάθε έκδοση (version) δομής inode στη μνήμη φλας διαθέτει έναν χαρακτηριστικό **αριθμό έκδοσης (version number)**. Οι αριθμοί έκδοσης, δεν επαναχρησιμοποιούνται ενώ δίνουν τη δυνατότητα στο σύστημα host να ανακατασκευάσει έναν απευθείας πίνακα αντιστοίχισης inode (direct inode map) από τον ανάστροφο πίνακα αντιστοίχισης (inverse map) που είναι αποθηκευμένος στη μνήμη φλας.

Στο JFFSv2, το log (ημερολόγιο) αποτελείται από μία συνδεδεμένη λίστα (linked list) με κόμβους μεταβλητού μεγέθους (variable-length nodes). Οι περισσότεροι κόμβοι περιέχουν μέρος των αρχείων καθώς και αντίγραφα των μεταδεδομένων αυτών των αρχείων. Υπάρχουν επίσης ειδικοί κόμβοι ευρετηρίου (directory-entry nodes) στους οποίους αποθηκεύονται το όνομα ενός αρχείου (file name) μαζί με τον αντίστοιχο αριθμό inode (inode number).

Κατά τη φάση προσάρτησης (at mount time), το σύστημα σαρώνει όλους τους κόμβους στο log (ημερολόγιο) και κατασκευάζει δύο δομές δεδομένων. Η πρώτη δομή δεδομένων είναι ένας απευθείας πίνακας αντιστοίχισης αριθμών inode στις πιο πρόσφατες εκδόσεις (versions) αυτών των αριθμών στη μνήμη φλας. Η αντιστοίχιση αυτή διατηρείται σε έναν **πίνακα κατακερματισμού (hash table)**. Η δεύτερη δομή δεδομένων είναι μία συλλογή δομών που αντιπροσωπεύουν κάθε έγκυρο κόμβο στη μνήμη φλας. Κάθε δομή συμμετέχει σε δύο συνδεδεμένες λίστες. Η μία συνδέει όλους τους κόμβους σύμφωνα με τις φυσικές διευθύνσεις και η άλλη περιλαμβάνει όλους τους κόμβους ενός αρχείου σε πλήρη διάταξη/ταξινόμηση. Η λίστα με τους κόμβους ενός αρχείου, συγκροτεί έναν απευθείας πίνακα αντιστοίχισης θέσεων του αρχείου (file positions) σε διευθύνσεις της μνήμης φλας. Επειδή, τόσο η αντιστοίχιση αριθμών inode σε διευθύνσεις της μνήμης φλας, όσο και η αντιστοίχιση θέσεων του αρχείου σε διευθύνσεις της μνήμης φλας, διατηρούνται αποκλειστικά στη μνήμη RAM, η δομή δεδομένων στη μνήμη φλας είναι πολύ απλή. Πιο συγκεκριμένα, όταν ένα αρχείο επεκτείνεται (extended) ή τροποποιείται, στη μνήμη φλας γράφονται μόνο τα νέα δεδομένα και ο αριθμός inode χωρίς καμία άλλη πληροφορία αντιστοίχισης (mapping information). Μία εμφανής συνέπεια της παραπάνω σχεδιαστικής επιλογής είναι βέβαια η υψηλή χρήση της μνήμης RAM.

3.3.2.2 Εξισορρόπηση φθοράς (Wear leveling)

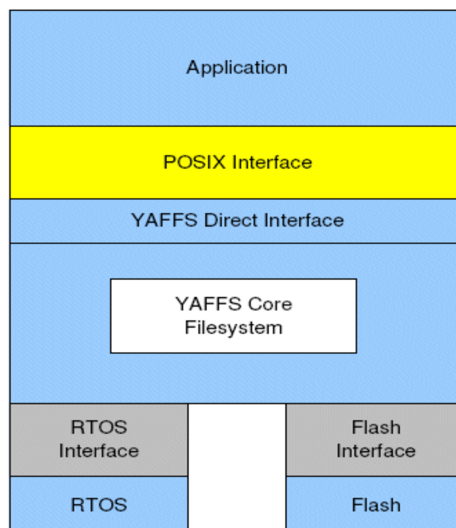
Το JFFSv2 χρησιμοποιεί μία απλή τεχνική εξισσορόπησης φθοράς (wear leveling technique). Πιο συγκεκριμένα, τις περισσότερες φορές, το **συστατικό εκκαθάρισης (cleaner)** επιλέγει για διαγραφή, μονάδες σβησίματος (erase units) οι οποίες περιέχουν τουλάχιστον κάποια απαρχαιωμένα δεδομένα (obsolete data) (το σχετικό άρθρο του JFFSv2 δεν προσδιορίζει επακριβώς τη στρατηγική επιλογής των μονάδων σβησίματος, αλλά πιθανόν η επιλογή βασίζεται στην ποσότητα απαρχαιωμένων δεδομένων σε κάθε μονάδα σβησίματος). Όμως, μετά από κάθε 100 λειτουργίες καθαρισμού (cleaning operations), το συστατικό καθαρισμού (cleaner) επιλέγει μία μονάδα σβησίματος που περιέχει μόνο έγκυρα δεδομένα σε μία προσπάθεια να διασκορπίσει τα στατικά δεδομένα σε όλο το εύρος της συσκευής μνήμης φλας.

Το συστατικό καθαρισμού, μπορεί να συγχωνεύσει μικρά μπλοκ δεδομένων που ανήκουν σε ένα αρχείο, σε ένα μεγάλο κομμάτι (chunk) μνήμης. Παρά το ότι κάτι τέτοιο γίνεται με σκοπό την αύξηση της απόδοσης, στην πραγματικότητα τα αποτελέσματα δεν είναι πάντοτε τα επιθυμητά. Πιο συγκεκριμένα, εάν τροποποιηθεί μόνο ένα μέρος από το μεγάλο κομμάτι, τότε θα γραφτεί στη μνήμη φλας ένα ολόκληρο νέο αντίγραφο του κομματιού (chunk), μιας και οι εγγραφές (writes) δεν επιτρέπεται να τροποποιούν μέρη έγκυρων κομματιών.

3.4 YAFFS: Yet Another Flash File System

Το YAFFS (Yet Another Flash File System) είναι ένα σύστημα αρχείων ειδικά σχεδιασμένο για χρήση σε **NAND μνήμη φλας**. Χαρακτηρίζεται από υψηλή μεταφερσιμότητα (highly portable) ενώ είναι συμβατό με λειτουργικά συστήματα όπως Linux, ucLinux και Windows CE. Η ακαταλληλότητα τόσο του JFFSv1 όσο και του JFFSv2 για NAND μνήμες φλας, οδήγησαν στη δημιουργία του YAFFS. Τα βασικά χαρακτηριστικά του YAFFS είναι τα παρακάτω:

- Ταχύτητα. Αρκετά γρηγορότερο από πολλά άλλα συστήματα αρχείων για μνήμη φλας.
- Εύκολα μεταφέρσιμο (easily ported). Είναι συμβατό με GNU/Linux, WinCE, eCOS (embedded Configurable operating systems), pSOS (plug-in Silicon Operating Systems) και VxWorks (Unix-like real-time operating system).
- Βασίζεται στην πολιτική των συστημάτων αρχείων με δομή log (log-structured file systems), ενώ παράλληλα παρέχει τεχνικές εξισορρόπησης φθοράς (wear leveling techniques).
- Υποστηρίζει διάφορους τύπους NAND μνήμης φλας όπως για παράδειγμα τσιπ NAND μνήμης φλας με σελίδα (page) 2KB καθώς και με σελίδα 512 Bytes.
- Παρέχει γρήγορη προσάρτηση (fast mount).
- Τυπικά, χρησιμοποιεί λιγότερη μνήμη RAM συγκριτικά με άλλα ανταγωνιστικά συστήματα αρχείων.



Σχήμα 3.1: Στρωματοποιημένη απεικόνιση του YAFFS.

3.4.1 Οργάνωση αποθήκευσης (Storage format)

Στο YAFFS, τα αρχεία (files) αποθηκεύονται σε κομμάτια (chunks) καθορισμένου μεγέθους. Το μέγεθος των κομματιών μπορεί να είναι 512 bytes, 1KB ή 2KB. Το σύστημα αρχείων συσχετίζει μία επικεφαλίδα (header) με κάθε κομμάτι. Η επικεφαλίδα είναι 16 bytes για κομμάτια μεγέθους 512 bytes, 30 bytes για κομμάτια μεγέθους 1KB και 42 bytes για κομμάτια 2KB. Κάθε αρχείο περιλαμβάνει ένα **κυρίαρχο κομμάτι (header chunk)**, στο οποίο αποθηκεύονται το όνομα του αρχείου (file's name), οι άδειες (permissions) πάνω στο αρχείο καθώς και τυχόν άλλα κομμάτια δεδομένων (data chunks). Όπως και στο JFFSv2, η μόνη πληροφορία αντιστοίχισης (mapping information) που αποθηκεύεται στη μνήμη φλας είναι το περιεχόμενο του κάθε κομματιού, το οποίο αποθηκεύεται ως μέρος της επικεφαλίδας. Αυτό έχει ως αποτέλεσμα, κατά τη φάση προσάρτησης (at mount time), να καθίσταται αναγκαία η ανάγνωση όλων των επικεφαλίδων από τη μνήμη φλας, για την κατασκευή του πίνακα αντιστοίχισης αναγνωριστικών αρχείων και του πίνακα αντιστοίχισης περιεχομένων των αρχείων, οι οποίοι πίνακες αποθηκεύονται στη συνέχεια στη μνήμη RAM.

Για την εξοικονόμηση μνήμης RAM, το YAFFS χρησιμοποιεί μία αποδοτική δομή αντιστοίχισης για να αντιστοιχίζει θέσεις αρχείων (file locations) σε φυσικές διευθύνσεις στη μνήμη φλας (physical flash addresses). Πιο συγκεκριμένα, χρησιμοποιείται μία δενδρική δομή (tree structure) με κόμβους των 32 bytes. Οι εσωτερικοί κόμβοι περιλαμβάνουν 8 δείκτες (pointers) προς άλλους κόμβους, ενώ οι κόμβοι φύλλα (leaf nodes) διαθέτουν 16 δείκτες των 2 bytes προς φυσικές διευθύνσεις. Σε μεγάλες μνήμες φλας, λέξεις (words) των 16 bits δεν μπορούν να δείχνουν (point) προς ένα αυθαίρετο (arbitrary) κομμάτι (chunk). Γι'αυτόν ακριβώς το λόγο, το YAFFS χρησιμοποιεί ένα σχήμα που ονομάζεται **προσεγγιστικοί δείκτες (approximate pointers)**. Η τιμή κάθε δείκτη αντιπροσωπεύει μία συνεχόμενη εμβέλεια κομματιών (contiguous range of chunks). Για παράδειγμα, εάν η μνήμη φλας περιέχει 2^{18} κομμάτια, τότε κάθε τιμή δείκτη των 16 bits αντιπροσωπεύει 4 διαφορετικά κομμάτια. Για την εύρεση του κομματιού που περιέχει το ζητούμενο μπλοκ δεδομένων, το σύστημα σαρώνει τις επικεφαλίδες των 4 κομματιών. Για την αποφυγή τροποποιήσεων της επικεφαλίδας του αρχείου, κάθε φορά που επιχειρείται μία πράξη, κάθε επικεφαλίδα κομματιού (chunk's header) περιέχει το ποσό των δεδομένων που αποθηκεύονται στο κομμάτι. Έτσι, μετά από μία πράξη, γράφεται ένα νέο κομμάτι «ουράς» (tail chunk). Το νέο μέγεθος της «ουράς» μαζί με τον αριθμό των γεμάτων (full) μπλοκ, δίνουν το μέγεθος του αρχείου.

3.4.2 YAFFSv1

Η πρώτη έκδοση του YAFFS (YAFFSv1), χρησιμοποιεί κομμάτια (chunks) μεγέθους 512 bytes. Η ανακήρυξη ενός κομματιού ως μη έγκυρο (invalid) γίνεται «καθαρίζοντας» (clear) (αλλαγή της τιμής των bits από 1 σε 0) ένα byte από την επικεφαλίδα του. Για να εξασφαλιστεί ότι τυχαία λάθη σε επίπεδο bit, τα οποία είναι αρκετά συχνά σε συσκευές NAND, δεν προκαλούν την επανεμφάνιση διαγραμμένων κομματιών ως έγκυρα (ή και το αντίστροφο), η μη εγκυρότητα (invalidity) σηματοδοτείτε από 4 ή και παραπάνω μηδενικά bits μέσα στο byte. Κάτι τέτοιο απαιτεί να γράφουμε δύο φορές σε κάθε σελίδα, προτού ανακτηθεί η μονάδα σβησίματος.

3.4.3 YAFFSv2

Το βασικό κίνητρο για το YAFFSv2 ήταν η υποστήριξη της νέας NAND μνήμης φλας με σελίδες 2KB αντί για 512 bytes, καθώς και η αυστηρά σειριακή σειρά εγγραφής σελίδας (strictly sequential page writing order). Για να επιτύχει τα παραπάνω, το YAFFSv2 χρησιμοποιεί έναν νέο σχεδιασμό ο οποίος επιφέρει τα παρακάτω πλεονεκτήματα:

- Η ανυπαρξία επανεγγραφών σελίδας (page rewrites) επιταχύνει όλες τις λειτουργίες/πράξεις (operations).
- Ικανότητα εκμετάλλευσης του ταυτόχρονου προγραμματισμού σελίδας σε κάποια τσιπ (chips) της μνήμης φλας.
- Βελτίωση της απόδοσης βασικών λειτουργιών/πράξεων σε σύγκριση με το YAFFSv1 (εγγραφή: 1.5x έως 5x, διαγραφή: 4x, συλλογή απορριμμάτων: 2x).
- Μικρότερη χρήση της μνήμης RAM σε σχέση με το YAFFSv1 (προσεγγιστικά 25% έως 50% μικρότερη χρήση RAM σε σχέση με το YAFFSv1).
- Υποστηρίζει MLC φλας μνήμη NAND των εταιριών Toshiba και SanDisk.

Το YAFFSv2, χρησιμοποιεί έναν πιο πολύπλοκο μηχανισμό για να ανακηρύσσει τα κομμάτια (chunks) μεγέθους 1KB ή 2KB ως μη έγκυρα. Ο βασικός στόχος του παραπάνω μηχανισμού είναι να επιτύχει μία αυστηρά σειριακή σειρά εγγραφής μέσα στις μονάδες σβησίματος, έτσι ώστε οι σβησμένες σελίδες (erased pages) να γράφονται η μία μετά την άλλη και να μην επανεγγράφονται ποτέ. Κάτι τέτοιο επιτυγχάνεται με τη χρήση δύο μηχανισμών. Πρώτον, κάθε επικεφαλίδα κομματιού, περιέχει όχι μόνο το αναγνωριστικό του αρχείου (file ID) αλλά και έναν αύξοντα αριθμό (sequence number). Με βάση τον αύξοντα αριθμό, αποφασίζεται ποιό είναι το έγκυρο κομμάτι από όλα τα κομμάτια που αντιπροσωπεύουν ένα μπλοκ ενός αρχείου. Τα υπόλοιπα κομμάτια μπορούν να ανακτηθούν (reclaimed). Δεύτερον, η διαγραφή αρχείων και καταλόγων (directories), γίνεται με την μετακίνησή τους σε έναν κατάλογο απορριμμάτων (trash directory), ο οποίος σημειώνει (marks) όλα τα κομμάτια τους ως υποψήφια για συλλογή απορριμμάτων (garbage collection). Όταν σβηστεί και το τελευταίο κομμάτι ενός διαγραμμένου αρχείου, τότε διαγράφεται από μόνο του και το αρχείο από τον κατάλογο απορριμμάτων (trash directory).

3.4.4 Εξισορρόπηση φθοράς (Wear leveling)

Η εξισορρόπηση φθοράς επιτυγχάνεται από την σποραδική τυχαία επιλογή μίας μονάδας σβησίματος για ανάκτηση. Όπως ακριβώς και στο JFFS, τις περισσότερες φορές επιλέγεται για σβήσιμο μία μονάδα σβησίματος που δεν περιέχει καθόλου έγκυρα δεδομένα.

3.4.5 Σύγκριση YAFFS και JFFSv2

Πρώτα, θα παρουσιάσουμε δύο βασικούς λόγους για τους οποίους το JFFS (JFFSv1, JFFSv2) δεν είναι κατάλληλο για συσκευές **NAND** μνήμης φλας:

- Τυπικά, οι NAND μνήμες φλας είναι πολύ μεγαλύτερες από τις NOR μνήμες φλας (για παράδειγμα, ένα τσιπ NAND μνήμης φλας των 128MB θεωρείται μεγάλο, ενώ αντίστοιχα στη NOR μνήμη φλας, ένα τσιπ της τάξης των 16MB θεωρείται μεγάλο). Συνεπώς, οι δομές δεδομένων σχετικές με την διαχείριση της μνήμης φλας, είναι πολύ μεγαλύτερες στις NAND από ότι στις NOR συσκευές. Επιπρόσθετα, στο JFFSv2, γίνεται σάρωση της μνήμης κατά την εκκίνηση (boot), για την δημιουργία των δομών δεδομένων που αναπαριστούν (represent) τους κόμβους του συστήματος. Κάτι τέτοιο αυξάνει το χρόνο εκκίνησης.

- Η NAND μνήμη φλας δεν είναι τυχαίας προσπέλασης (random access), αλλά αντιθέτως πρέπει να διαβάζεται σειριακά. Αυτό πρακτικά σημαίνει ότι η διαδικασία σάρωσης πρέπει να διαβάζει ολόκληρη τη μνήμη φλας, με αποτέλεσμα να επιβραδύνεται τόσο η διαδικασία σάρωσης (scanning) όσο και η διαδικασία εκκίνησης (booting).

Παρακάτω, ακολουθεί η σύγκριση μεταξύ του YAFFS και του JFFSv2:

1. Το YAFFS χρησιμοποιεί λιγότερη μνήμη για να διατηρεί την κατάστασή του με αποτέλεσμα να κλιμακώνει (scales) καλύτερα από το JFFSv2.

2. Ο μηχανισμός συλλογής απορριμμάτων του YAFFS είναι απλούστερος και ταχύτερος από αυτόν του JFFSv2.

3. Το YAFFS χρησιμοποιεί μία ολόκληρη σελίδα ανά αρχείο, αποκλειστικά για επικεφαλίδες (headers), ενώ δεν παρέχει μηχανισμούς συμπίεσης (compression). Αυτό έχει ως αποτέλεσμα, το JFFSv2 να είναι το καταλληλότερο για μικρές διαμερίσεις (partitions).

4. Αν το μέγεθος των διαμερίσεων είναι μικρότερο των 64MB, τότε το JFFSv2 είναι η καλύτερη επιλογή. Όσο όμως μεγαλώνει το μέγεθος της διαμέρισης, συνίσταται η χρήση του YAFFS.

3.5 Trimble File System

Το Trimble σύστημα αρχείων (Trimble File System), σχεδιάστηκε από τους Marshall και Manning της εταιρίας Trimble Navigation, αποκλειστικά για χρήση σε **NOR μνήμη φλας**. Ο Manning συμμετείχε επίσης στην δημιουργία του YAFFS που εξετάσαμε παραπάνω. Στις παρακάτω ενότητες, παρουσιάζονται συνοπτικά, η βασική οργάνωση αποθήκευσης (storage format) που ακολουθεί το Trimble σύστημα αρχείων καθώς και ο τρόπος ανάκτησης μονάδων σβησίματος (erase units reclamation).

3.5.1 Οργάνωση αποθήκευσης (Storage format)

Η όλη δομή (overall structure) του Trimble συστήματος αρχείων, μοιάζει αρκετά με αυτή του YAFFS. Πιο συγκεκριμένα, τα αρχεία σπάνε σε κομμάτια (chunks) των 252 byte, ενώ κάθε κομμάτι (chunk) αποθηκεύεται με μία επικεφαλίδα (header) των 4 byte, σε ένα τομέα (sector) των 256 byte της μνήμης φλας. Η επικεφαλίδα των 4 byte περιλαμβάνει τον αριθμό του αρχείου (file number) καθώς και τον αριθμό του κομματιού (chunk number) του αρχείου. Κάθε αρχείο, έχει έναν κυρίαρχο τομέα (header sector) ο οποίος περιέχει τον αριθμό του αρχείου (files number), μία έγκυρη/μη έγκυρη λέξη (valid/invalid word), ένα όνομα αρχείου (file name) και μέχρι 14 εγγραφές αρχείου (file records), εκ των οποίων μόνο μία και συγκεκριμένα η τελευταία, είναι έγκυρη (valid). Κάθε μία από τις 14 εγγραφές περιλαμβάνει το μέγεθος του αρχείου (file's size), μία μεταβλητή για έλεγχο αθροίσματος (checksum) καθώς και τον χρόνο τελευταίας τροποποίησης του αρχείου (last modification time). Όποτε ένα αρχείο τροποποιείται, χρησιμοποιείται μία νέα εγγραφή, και όταν και οι 14 εγγραφές χρησιμοποιούνται, τότε ανατίθεται στο αρχείο ένας νέος κυρίαρχος τομέας (header sector).

Όπως ακριβώς και στο YAFFS, όλη η πληροφορία αντιστοίχισης (mapping information) αποθηκεύεται στη RAM μιάς και η μνήμη φλας δεν περιέχει άλλη δομή αντιστοίχισης (mapping structure) πέραν του ανάστροφου πίνακα αντιστοίχισης (inverse map table).

3.5.2 Ανάκτηση μονάδων σβησίματος (erase units reclamation)

Η επιλογή των μονάδων σβησίματος (erase units) για ανάκτηση (reclamation), βασίζεται αποκλειστικά στον αριθμό των έγκυρων τομέων (valid sectors) που αυτές περιέχουν και πραγματοποιείται μόνο αν περιλαμβάνουν δεσμευμένους τομείς (no free sectors). Για την αποφυγή της απώλειας του **μετρητή σβησίματος (erase counter),** κατά τη διάρκεια μίας κατάστασης κατάρρευσης (crash), η οποία εμφανίζεται αμέσως μετά το σβήσιμο μίας μονάδας αλλά πριν την εγγραφή της επικεφαλίδας της, ο μετρητής σβησίματος γράφεται εκ των προτέρων σε έναν ειδικό τομέα (special sector). Η ανάθεση των τομέων (sectors) στις μονάδες σβησίματος γίνεται σειριακά ενώ η επιλογή της επόμενης μονάδας σβησίματος προς χρήση βασίζεται στους μετρητές σβησίματος (erase counters), παρέχοντας με αυτόν τον τρόπο κάποια εξισορρόπηση φθοράς (wear leveling) μεταξύ των μπλοκ της μνήμης φλας.

3.6 Research-In-Motion File System

Η εταιρία Research In Motion, ειδική στην κατασκευή συσκευών παλάμης (handheld devices) και έξυπνων κινητών τηλεφώνων (smart mobile phones), δημιούργησε ένα σύστημα αρχείων βασισμένο στη δομή log (ημερολογίου) (log-structured file system) ειδικό για μνήμες φλας. Το σύστημα αρχείων είναι σχεδιασμένο να αποθηκεύει συνεχόμενες εγγραφές δεδομένων μεταβλητού μεγέθους, κάθε μία με ένα μοναδικό αναγνωριστικό (unique identifier).

3.6.1 Οργάνωση αποθήκευσης (Storage format)

Στο Research In Motion σύστημα αρχείων, η μνήμη φλας χωρίζεται (partitioned) σε μία περιοχή για τα προγράμματα και σε μία περιοχή για το σύστημα αρχείων. Ο διαχωρισμός αυτός γίνεται για να εκτελούνται τα προγράμματα εντός θέσης (in-place) απευθείας από την NOR μνήμη φλας. Η περιοχή του συστήματος αρχείων οργανώνεται σαν ένα κυκλικό ημερολόγιο (circular log) το οποίο περιλαμβάνει μία αλληλουχία εγγραφών (records). Κάθε εγγραφή ξεκινάει με μία επικεφαλίδα (header) η οποία περιέχει, το μέγεθος της εγγραφής (record's size), ένα αναγνωριστικό (identity), μία σημαία μη εγκυρότητας (invalidation flag) καθώς και έναν δείκτη προς την επόμενη εγγραφή του ίδιου αρχείου.

Εξαιτίας της κυκλικής δομής του log (ημερολογίου), η διαδικασία καθαρισμού (cleaning process) μπορεί να μην είναι αρκετά αποδοτική. Πιο συγκεκριμένα, η παλαιότερη μονάδα σβησίματος μπορεί να μην περιέχει αρκετά απαρχαιωμένα δεδομένα (obsolete data), αλλά παρ'όλα αυτά θα διαγραφεί/σβηστεί (cleaned). Για να αντιμετωπίσει αυτή τη κατάσταση, το Research In Motion σύστημα αρχείων διαμερίζει (partition) επιπλέον τη μνήμη φλας σε ένα log για τα λεγόμενα «καυτά» (hot) (συχνά τροποποιούμενα) δεδομένα και σε ένα log για τα «κρύα» (cold) (σπάνια τροποποιούμενα) δεδομένα.

Διατηρώντας συνεχόμενες τις εγγραφές, εξασφαλίζουμε στο σύστημα αρχείων τη δυνατότητα να επιστρέφει τους δείκτες απευθείας στη NOR μνήμη φλας κατά τη λειτουργία ανάγνωσης (reading operation). Προφανώς, κάτι τέτοιο είναι εφικτό μέσω μίας API (Application Programming Interface) η οποία κάθε χρονική στιγμή επιτρέπει πρόσβαση σε μία μόνο εγγραφή και όχι σε πολλαπλές εγγραφές. Τέλος, ένας απευθείας πίνακας αντιστοίχισης, ο οποίος αντιστοιχίζει λογικούς αριθμούς εγγραφών (logical record numbers) σε θέσεις της μνήμης φλας, διατηρείται στη μνήμη RAM.

Βιβλιογραφία κεφαλαίου

[1] Eran Gal and Sivan Toledo. Algorithms and Data Structures for Flash Memories. (χρήση στις σελίδες 31-33, 36-38, 39-40, 41-43)

[2] David WoodHouse. JFFS: The Journalling Flash File System. Red Hat, Inc. (χρήση στις σελίδες 34-38)

[3] www.yaffs.net. (χρήση στις σελίδες 38, 41)

Κεφάλαιο 4

Δομές δεδομένων ευρετηρίου (Indexing data structures)

Περιεχόμενα

4.1 Εισαγωγή……………………………………………………………….46

4.2 Το στρώμα BFTL……………………………………………….........47

4.2.1 Λόγοι που οδήγησαν στην δημιουργία του BFTL………………………...47

4.2.2 Ο σχεδιασμός και η υλοποίηση του BFTL……………………………...49

4.2.2.1 Γενική επισκόπηση του BFTL……………………………………….......50

4.2.2.2 Φυσική αναπαράσταση κόμβου B-δένδρου: Μονάδες ευρετηρίου……….51

4.2.2.3 Πολιτική μεταφοράς (Commit Policy)……………………………………52

4.2.2.4 Ο πίνακας μετάφρασης κόμβων (Node translation table)……………….....54

4.2.3 Ανάλυση πολυπλοκότητας του BFTL……………………………………..56

4.2.4 Εκτίμηση της απόδοσης του BFTL………………………………………...58

4.2.4.1 Επισκόπηση πειραμάτων.. 58

4.2.4.2 Απόδοση δημιουργίας B-δένδρων.. 59

4.2.4.3 Απόδοση διατήρησης B-δένδρων... 61

4.2.4.4 Απόδοση αναζήτησης σε B-δένδρα... 62

4.2.4.5 Κατανάλωση ενέργειας... 63

4.2.4.6 Απόδοση BFTL για διάφορες τιμές του c 63

4.3 B^+ - δένδρα...64

4.3.1 Κατηγορίες B^+ - δένδρων για μνήμη φλας...66

4.3.2 Χρήση B^+ - δένδρων στο JFFS…………………………………………66

4.4 μ - δένδρα………………………………………………………………..67

4.4.1 Ανασκόπηση των μ - δένδρων..67

4.4.2 Διαδικασία ανάκτησης στοιχείου σε μ - δένδρο......................................69

4.4.3 Διαδικασία εισαγωγής στοιχείου σε μ - δένδρο......................................70

4.4.4 Διαδικασία διαγραφής στοιχείου σε μ - δένδρο......................................72

4.4.5 Ανάλυση πολυπλοκότητας μ - δένδρου...74

4.4.5.1 Κόστος πράξεων.. 74

4.4.5.2 Το ύψος των δένδρων...74

4.4.5.3 Χωρικός φόρτος (Space overhead)……………………………………...76

4.4.6 Αποτελέσματα πειραμάτων...77

4.5 FlashDB………………………………………………………………78

4.5.1 Αυτορυθμιζόμενη ευρετηρίαση (Self-tuning indexing)…………………...79

4.5.1.1 Σχεδίαση B^+ - δένδρων (ST)...79

4.5.1.1.1 Συστατικά διαχειριστή αποθήκευσης..80

4.5.1.1.2 Βασικές πράξεις...81

4.5.1.1.3 Δομή τομέα, NTT και καταχωρίσεων ημερολογίου...................................82

4.5.1.2 Βασικά ζητήματα αυτορύθμισης..83

4.5.1.2.1 Αλγόριθμος εναλλαγής κατάστασης..83

4.5.1.3 Βελτιστοποίηση B^+ - δένδρων (ST)..85

4.5.1.3.1 Συμπίεση ημερολογίου και συλλογή απορριμμάτων..................................85

4.5.1.3.2 Μεγαλύτερη περιοχή προσωρινής αποθήκευσης......................................86

4.5.2 Εκτίμηση της απόδοσης της FlashDB……………………………………..86
Βιβλιογραφία κεφαλαίου...88

4.1 Εισαγωγή

Η μνήμη φλας έχει καθιερωθεί πλέον ως ένα δημοφιλές εναλλακτικό μέσο για τον σχεδιασμό **συστημάτων αποθήκευσης (storage systems)**, εξαιτίας βέβαια των βασικών χαρακτηριστικών που τη διέπουν. Αυτά τα χαρακτηριστικά περιλαμβάνουν, την **αντοχή της μνήμης φλας σε δονήσεις (shock-resistant)**, τις **επιδόσεις κατανάλωσης ενέργειας (energy-efficient)** καθώς και τη **μη ευμετάβλητη φύση (non-volatile nature)** της. Τα τελευταία χρόνια, σημειώνονται μεγάλες τεχνολογικές τομές σχετικά με τις μνήμες φλας, τόσο σε επίπεδο χωρητικότητας (capacity) όσο και σε επίπεδο αξιοπιστίας (reliability). Σε πολλές εφαρμογές, η μνήμη φλας έχει αντικαταστήσει την χρήση σκληρών δίσκων (hard disks).

Η υλοποίηση των **δομών δεδομένων ευρετηρίου (index data structures)**, οι οποίες είναι πολύ δημοφιλείς στην οργάνωση των δεδομένων των σκληρών δίσκων, πρέπει να γίνεται λαμβάνοντας υπόψιν και την μνήμη φλας. Καθώς η μνήμη φλας καθιερώνεται όλο και περισσότερο ως αποθηκευτικό μέσο σε πολλά ενσωματωμένα συστήματα, πολλά συστήματα αρχείων καθώς και συστήματα διαχείρισης βάσεων δεδομένων (database management systems), «οικοδομούνται» (built) πάνω της. Τα συστήματα αυτά, απαιτούν μία αποδοτική δομή ευρετηρίασης (efficient index structure), η οποία θα εντοπίζει ταχύτατα ένα συγκεκριμένο αντικείμενο (item) από μία μεγάλη ποσότητα καταχωρίσεων καταλόγου (directory entries) ή εγγραφών μίας βάσης δεδομένων (database records). Παρ'όλα αυτά, εξαιτίας των χαρακτηριστικών της μνήμης φλας, οι **παραδοσιακοί σχεδιασμοί** των δομών ευρετηρίου μειώνουν την αξιοπιστία της μνήμης φλας, γεγονός που επιδρά σημαντικά στην μείωση της απόδοσης των φλας- συστημάτων αποθήκευσης (flash-memory storage systems).

Υπάρχουν δύο βασικοί λόγοι οι οποίοι επιδρούν αρνητικά στην αποδοτικότητα (efficiency) των παραδοσιακών δομών ευρετηρίου (index structures) κατά τη χρήση τους σε μνήμη φλας:

1. Η μονή εγγραφή με διαγραφές/σβησίματα μεγάλου όγκου (write-once with bulk erases).
2. Η ανθεκτικότητα (endurance) της μνήμης φλας.

Όπως είναι γνωστό και από το 1^{ο} κεφάλαιο, η μνήμη φλας δεν μπορεί να πανωγραφτεί/ανανεωθεί (overwritten/updated) αν πρώτα δεν υποστεί διαγραφή. Ως αποτέλεσμα του παραπάνω περιορισμού, είναι η συνύπαρξη στη μνήμη φλας, μη έγκυρων εκδόσεων (invalid versions) δεδομένων μαζί με τις τελευταίες εκδόσεις (latest versions) αυτών των δεδομένων. Επιπλέον, οι συχνές διαγραφές συγκεκριμένων θέσεων (locations) της μνήμης φλας, μειώνουν γρήγορα την διάρκεια ζωής (lifetime) της μνήμης, μιάς και κάθε μονάδα σβησίματος έχει ένα περιορισμό στον πλήθος των διαγραφών που μπορεί να υποστεί (limited cycle count on erase operations).

Ως αποτέλεσμα των παραπάνω, κρίνεται ως απαραίτητη η δημιουργία δομών δεδομένων ευρετηρίου (index data structures) κατάλληλες για χρήση σε μνήμες φλας. Σε αυτό το κεφάλαιο, θα παρουσιαστούν λεπτομερώς ορισμένες σημαντικές δομές ευρετηρίου, οι οποίες λαμβάνουν υπόψιν τους τα χαρακτηριστικά και τους παραπάνω περιορισμούς που επιβάλλει η μνήμη φλας, με αποτέλεσμα να αυξάνουν την απόδοση των συστημάτων στα οποία χρησιμοποιούνται.

4.2 Το στρώμα BFTL

Στην ενότητα αυτή θα ασχοληθούμε με την παρουσίαση του **BFTL** (ένα αποδοτικό στρώμα B-δένδρου για φλας-συστήματα αποθήκευσης-an efficient B-tree layer for flash-memory storage systems). Το BFTL ενοποιεί τη δομή ευρετηρίου B-δένδρο (B-tree index structure) με τους μηχανισμούς που παρέχει το FTL (Flash Translation Layer). Πρόκειται για ένα στρώμα (layer) που διαχειρίζεται τα B-δένδρα των φλας-συστημάτων αποθήκευσης (flash-memory storage systems), ενώ παράλληλα χειρίζεται τις εντατικές λειτουργίες επιπέδου byte (byte-wise operations) που οφείλονται στις προσπελάσεις των B-δένδρων. Το BFTL εισάγεται σαν στρώμα **ανάμεσα στο σύστημα αρχείων (file system) και το FTL (Flash Translation Layer)**. Μπορεί να προσαρμοστεί (adopted) ή να αφαιρεθεί ανάλογα με τις απαιτήσεις του εκάστοτε συστήματος, ενώ βασικός του στόχος είναι η κατάλληλη υλοποίηση των B-δένδρων η οποία θα μειώσει τον φόρτο που προκαλείται από τον χειρισμό των B-δένδρων. Η υλοποίηση του BFTL γίνεται διαφανώς πάνω από το FTL, με αποτέλεσμα να μη χρειάζεται καμία τροποποίηση σε υπάρχουσες εφαρμογές που σχετίζονται με B-δένδρα (B-tree related applications).

Ο φόρτος των εντατικών λειτουργιών επιπέδου byte (byte-wise operations), προκαλείται από την εισαγωγή καταχωρίσεων (record inserting), την διαγραφή καταχωρίσεων (record deleting) καθώς και από την αναδιοργάνωση των B-δένδρων (B-tree reorganizing). Για παράδειγμα, η εισαγωγή μίας καταχώρισης έχει ως αποτέλεσμα την εισαγωγή ενός δείκτη δεδομένων (data pointer) σε ένα **κόμβο φύλλο (leaf node)**, και την πιθανή εισαγωγή δεικτών δένδρου (tree pointers) στο B-δένδρο. Τέτοιες ενέργειες, προκαλούν την αντιγραφή δεδομένων μεγάλης ποσότητας (συγκεκριμένα την αντιγραφή των αμετάβλητων δεδομένων και των δεικτών προς τους σχετιζόμενους κόμβους), εξαιτίας βέβαια των ανανεώσεων εκτός θέσης (out-place updates) που διέπουν τη μνήμη φλας. Το BFTL βελτιώνει σημαντικά την απόδοση του συστήματος ενώ παράλληλα μειώνει το φόρτο διαχείρισης της μνήμης φλας καθώς και την κατανάλωση ενέργειας, όταν υιοθετούνται από τη μνήμη φλας δομές δεδομένων ευρετηρίου. Στο σημείο αυτό αξίζει να αναφέρουμε ότι το BFTL μπορεί προσαρμοστεί και στα συστήματα αρχείων ειδικά για μνήμη φλας (όπως π.χ το YAFFS ή το JFFS) που μελετήσαμε στο προηγούμενο κεφάλαιο.

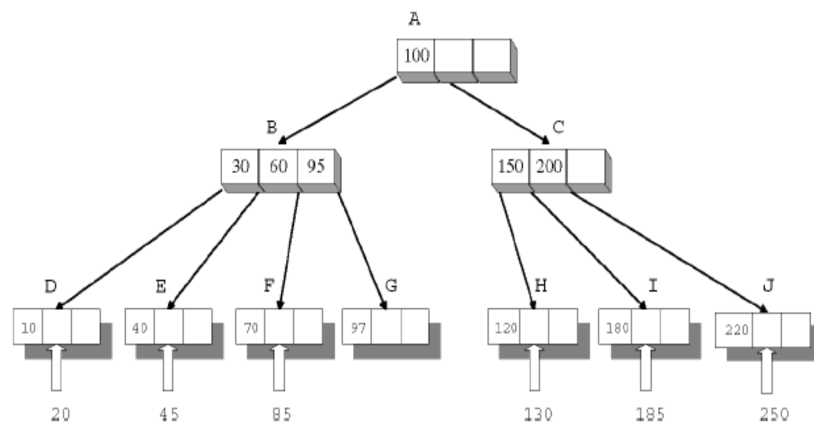
4.2.1 Λόγοι που οδήγησαν στη δημιουργία του BFTL

Καθώς η χωρητικότητα αποθήκευσης (storage capacity) της μνήμης φλας αυξάνεται διαρκώς, πολλά συστήματα όπως, κινητά τηλέφωνα (mobile phones), PDA’s (Personal Digital Assistants) καθώς και έξυπνα τηλέφωνα (smart phones), χρησιμοποιούν τη μνήμη φλας σαν δευτερεύουσα (επιπρόσθετη) αποθηκευτική συσκευή. Η δομή δεδομένων των B-δένδρων χρησιμοποιείται ευρέως σε εφαρμογές βάσεων δεδομένων καθώς και σε συστήματα βάσεων δεδομένων όπως το Oracle 8 και το SQL server της Microsoft. Εφαρμογές με ανάγκες επεξεργασίας σε επίπεδο αρχείων (file-processing needs), διάσημα συστήματα αρχείων με ημερολόγιο (journaling file systems) (όπως το XFS και το JFS) καθώς και πολλές άλλες εφαρμογές, που στο παρελθόν έτρεχαν πάνω από σκληρούς δίσκους, μπορεί τώρα να προσπελαύνουν δεδομένα που αποθηκεύονται στη μνήμη φλας. Εξαιτίας του τεράστιου φόρτου που προκαλείται από το χειρισμό των B-δένδρων στη μνήμη φλας, πολλές εφαρμογές που υιοθετούν B-δένδρα, αντιμετωπίζουν σοβαρά προβλήματα

απόδοσης κατά την προσπέλαση δεδομένων που βρίσκονται στη μνήμη φλας. Το παραπάνω πρόβλημα αποτελεί το βασικότερο κίνητρο της δημιουργίας του BFTL.

Τα B-δένδρα αποτελούν μία ιεραρχική δομή δεδομένων. Παρέχουν αποδοτικές λειτουργίες για την εισαγωγή (insert), διαγραφή (delete), εύρεση (find) και διαπέραση (traverse) των δεδομένων. Υπάρχουν δύο κατηγορίες κόμβων στα B-δένδρα: οι εσωτερικοί κόμβοι (internal nodes) και οι κόμβοι φύλλα (leaf nodes). Ένας εσωτερικός κόμβος, αποτελείται από μία πλήρως διατεταγμένη κατά αύξουσα σειρά λίστα με τιμές κλειδιών (key values) καθώς και συνδετικούς δείκτες (linkage pointers) προς τα παιδιά αυτού του κόμβου. Τα δεδομένα που αποθηκεύονται στο αριστερό υποδένδρο ενός κόμβου-πατέρα, έχουν πάντοτε κλειδιά με μικρότερη τιμή από αυτή του κόμβου-πατέρα, ενώ τα δεδομένα που αποθηκεύονται στο δεξιό υποδένδρο του κόμβου-πατέρα έχουν κλειδιά με μεγαλύτερη τιμή από αυτή του κόμβου πατέρα. Ένας κόμβος φύλλο ενός B-δένδρου περιέχει, μία τιμή κλειδιού και τον αντίστοιχο δείκτη εγγραφής (record pointer). Στις περισσότερες περιπτώσεις, τα B-δένδρα χρησιμοποιούνται σαν εξωτερικές (εκτός RAM) δομές δεδομένων ευρετηρίου, με σκοπό τη διατήρηση μεγάλου όγκου δεδομένων. Παραδοσιακά, τα B-δένδρα υλοποιούνται σε αποθηκευτικά συστήματα δίσκων (disk-storage systems) για την ελάττωση των λειτουργιών I/O. Παρ’όλα αυτά, μία απευθείας υιοθέτηση/εφαρμογή των B-δένδρων σε φλας-συστήματα αποθήκευσης (flash-memory storage systems), θα είχε ως αποτέλεσμα την αύξηση του φόρτου κατά τη διαχείριση της μνήμης φλας.

Θα παρουσιάσουμε πρώτα κάποιες συνηθισμένες λειτουργίες που πραγματοποιούνται σε ένα B-δένδρο.



Σχήμα 4.1: Μία δομή δεδομένων B-δένδρου.

Το σχήμα 4.1 παρουσιάζει ένα B-δένδρο όπου ο μέγιστος αριθμός παιδιών/μεγιστος βαθμός εξόδου ανά κόμβο είναι τέσσερα (fanout=4). Ας υποθέσουμε ότι θέλουμε να εισάγουμε στο παραπάνω δένδρο έξι διαφορετικές καταχωρίσεις (records), όπου τα πρωτεύοντα κλειδιά (primary keys) των καταχωρίσεων είναι 20, 45, 85, 130, 185 και 250 αντίστοιχα. Όπως φαίνεται και στο παραπάνω σχήμα, η πρώτη, η δεύτερη, η τρίτη, η τέταρτη, η πέμπτη και η έκτη καταχώριση, εισάγονται διαδοχικά στους κόμβους D, E, F, H, I και J. Παρατηρούμε λοιπόν ότι τροποποιούνται έξι διαφορετικοί κόμβοι του δένδρου. Ας επικεντρωθούμε τώρα στα αρχεία των δομών ευρετηρίου, μιάς και η αποθήκευση των δομών ευρετηρίου γίνεται ξεχωριστά από αυτή των καταχωρίσεων. Αν υποθέσουμε ότι κάθε

κόμβος του B-δένδρου αποθηκεύεται σε μία ξεχωριστή σελίδα, τότε απαιτούνται έξι εγγραφές σελίδων (page writes) για την ολοκλήρωση της παραπάνω εισαγωγής. Αν χρειαστεί να πραγματοποιηθούν και επαναζυγιστικές πράξεις (rebalancing operations), τότε απαιτούνται περισσότερες ανανεώσεις (updates) σε επίπεδο εσωτερικών κόμβων.

Η ανανέωση (update) ή η εγγραφή (write) δεδομένων στη μνήμη φλας είναι πολύ περίπλοκες και ακριβές λειτουργίες, σε σύγκριση με την εκτέλεση αυτών των λειτουργιών σε σκληρούς δίσκους. Από τη στιγμή που η μνήμη φλας υιοθετεί την τεχνική της **ανανέωσης εκτός θέσης (outplace update)**, μία ολόκληρη σελίδα των 512 bytes, η οποία περιέχει τη νέα έκδοση των δεδομένων, εγγράφεται στη μνήμη φλας, ενώ τα παλαιά δεδομένα (previous data) πρέπει να ανακηρυχτούν ως μη έγκυρα (invalid). Η εγγραφή σε επίπεδο σελίδας εισάγει μία αλληλουχία αρνητικών επιπτώσεων. Πιο συγκεκριμένα, ο ελεύθερος χώρος της μνήμης φλας καταναλώνεται πολύ γρήγορα με αποτέλεσμα οι μηχανισμοί συλλογής απορριμμάτων (garbage collection mechanisms) να ενεργοποιούνται πολύ συχνά για να ανακτήσουν ελεύθερο χώρο. Επιπρόσθετα, επειδή η μνήμη φλας σβήνεται πολύ συχνά (frequently erased), η διάρκεια ζωής της μειώνεται. Ένα άλλο βασικό πρόβλημα είναι αυτό της κατανάλωσης ενέργειας. Οι ανανεώσεις εκτός θέσης οδηγούν στην εφαρμογή των μηχανισμών συλλογής απορριμμάτων, οι οποίοι διαβάζουν σελίδες, γράφουν σελίδες και σβήνουν μπλοκ.

	Page Read 512 B	Page Write 512 B	Block Erase 16 KB
Performance(μs)	348	909	1,881
Energy Consumption($\mu joule$)	99	237.6	422.4

Πίνακας 4.1: Απόδοση μίας τυπικής NAND μνήμης φλας.

Επειδή οι εγγραφές και τα σβησίματα σελίδων καταναλώνουν πολύ περισσότερη ενέργεια απ'ότι οι αναγνώσεις σελίδων, γεγονός που μαρτυράται και από τον παραπάνω πίνακα, οι ανανεώσεις εκτός θέσης καταναλώνουν τελικά μεγάλες ποσότητες ενέργειας. Σε φορητές συσκευές (portable devices), όπου η ποσότητα ενέργειας που παρέχεται από τις μπαταρίες είναι περιορισμένη, η εξοικονόμηση ενέργειας (energy-saving) είναι σημαντικότατος παράγοντας.

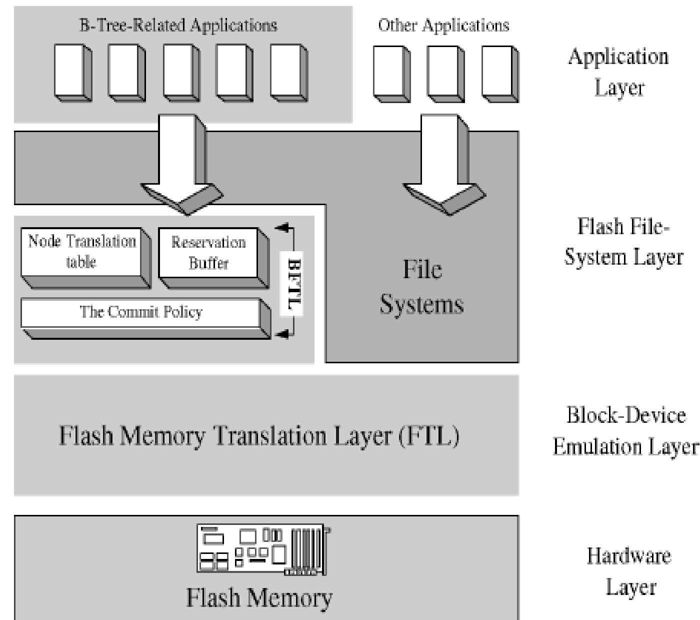
Όλα τα προβλήματα που αναφερθήκαν παραπάνω, οδήγησαν στη δημιουργία του BFTL, το οποίο όχι μόνο παρέχει μία συμβατή λύση σε υπάρχοντα συστήματα, αλλά πρόκειται για ένα στρώμα (layer) που μειώνει σημαντικά το φόρτο που προκαλείται από το χειρισμό των B-δένδρων.

4.2.2 Ο σχεδιασμός και η υλοποίηση του BFTL

Στις ενότητες που ακολουθούν, θα παρουσιαστεί το στρώμα B-δένδρου για φλας-συστήματα αποθήκευσης (BFTL), το οποίο έχει ως κύριο στόχο να ελαττώσει τις περιττές εγγραφές δεδομένων, οι οποίες οφείλονται στους hardware περιορισμούς της NAND μνήμης φλας. Πιο συγκεκριμένα, παρουσιάζεται η αρχιτεκτονική ενός συστήματος που υιοθετεί το BFTL, καθώς και οι λειτουργίες των βασικών συστατικών του BFTL.

4.2.2.1 Γενική επισκόπηση του BFTL

Το BFTL αφοσιώνεται κυρίως στην αποδοτική υλοποίηση δομών δεδομένων ευρετηρίου B-δένδρων πάνω από το FTL, ενώ παράλληλα παρέχει λειτουργίες επιπέδου συστήματος αρχείων για τη δημιουργία και διατήρηση B-δένδρων. Το BFTL είναι μέρος του λειτουργικού συστήματος.



Σχήμα 4.2: Αρχιτεκτονική συστήματος που χρησιμοποιεί το BFTL.

Όπως φαίνεται και στο παραπάνω σχήμα, το BFTL αποτελείται από μία **ειδική περιοχή προσωρινής αποθήκευσης (reservation buffer)** και από έναν **πίνακα μετάφρασης κόμβων (node translation table)**. Οι υπηρεσίες ευρετηρίου B-δένδρων που αιτούνται από τις εφαρμογές (application layer), διαχειρίζονται και μεταφράζονται από το σύστημα αρχείων στο BFTL, ενώ οι αιτήσεις μπλοκ συσκευών (block-device requests) στέλνονται από το BFTL στο FTL. Όταν μία εφαρμογή εισάγει, διαγράφει ή τροποποιεί καταχωρίσεις/εγγραφές (records), οι καινούργιες αυτές καταχωρίσεις (συχνά αποκαλούνται **«βρώμικες» καταχωρίσεις - dirty records**) αποθηκεύονται προσωρινά στην ειδική περιοχή προσωρινής αποθήκευσης (reservation buffer) του BFTL. Από τη στιγμή που η ειδική περιοχή προσωρινής αποθήκευσης έχει περιορισμένη χωρητικότητα, οι «βρώμικες» καταχωρίσεις πρέπει να μεταφέρονται εγκαίρως στη μνήμη φλας. Στο σημείο αυτό αξίζει να αναφέρουμε ότι, οι διαγραφές των καταχωρίσεων πραγματοποιούνται προσθέτοντας καταχωρίσεις ‘‘μη εγκυρότητας’’ (invalidation records) στην ειδική περιοχή προσωρινής αποθήκευσης.

Για τη μεταφορά των «βρώμικων» καταχωρίσεων (dirty records) της ειδική περιοχής προσωρινής αποθήκευσης, στη μνήμη φλας, το BFTL κατασκευάζει μία **μονάδα ευρετηρίου (index unit)** για κάθε «βρώμικη» καταχώριση. Οι μονάδες ευρετηρίου, απεικονίζουν/αντανακλούν τις εισαγωγές και διαγραφές πρωτεύοντος κλειδιού στο B-δένδρο, οι οποίες προκαλούνται από τις «βρώμικές» καταχωρίσεις. Η αποθήκευση των μονάδων ευρετηρίου καθώς και των «βρώμικων» καταχωρίσεων, γίνεται με δύο διαφορετικούς τρόπους. Πιο συγκεκριμένα, οι «βρώμικες»

καταχωρίσεις γράφονται (ή ανανεώνονται) στις ανατεθείσες (ή στις αρχικές-original) θέσεις. Από την άλλη πλευρά, εξαιτίας του μικρού μεγέθους της μονάδας ευρετηρίου (συγκριτικά με το μέγεθος σελίδας), η αποθήκευσή της αναλαμβάνεται από μία πολιτική μεταφοράς (commit policy). Ειδικότερα, πολλές μονάδες ευρετηρίου τοποθετούνται αποδοτικά σε λίγους τομείς (sectors), με σκοπό τη μείωση των εγγραφών σε επίπεδο σελίδων (page writes). Πιο συγκεκριμένα, η τοποθέτηση μονάδων ευρετηρίου που ανήκουν σε διαφορετικούς κόμβους του B-δένδρου, σε έναν μικρό αριθμό τομέων, ναι μεν ελαττώνει τον αριθμό των ανανεώσεων σε επίπεδο τομέων, αλλά οι μονάδες ευρετηρίου ενός κόμβου διασκορπίζονται σε διαφορετικούς τομείς. Για αυτόν ακριβώς το λόγο, το BFTL χρησιμοποιεί έναν πίνακα μετάφρασης κόμβων (node translation table), για να εντοπίζει με ευκολία τις μονάδες ευρετηρίου που ανήκουν σε έναν κόμβο.

4.2.2.2 Φυσική αναπαράσταση κόμβου B-δένδρου: Μονάδες ευρετηρίου

Όπως αναφέραμε και παραπάνω, το BFTL κατασκευάζει μία **μονάδα ευρετηρίου (index unit)** για κάθε «βρώμικη» καταχώριση. Οι μονάδες ευρετηρίου, απεικονίζουν/αντανακλούν τις εισαγωγές και διαγραφές πρωτεύοντος κλειδιού στο B-δένδρο, οι οποίες προκαλούνται από τις «βρώμικές» καταχωρίσεις. Με άλλα λόγια, μεταχειριζόμαστε κάθε μονάδα ευρετηρίου σαν μία τροποποίηση του αντίστοιχου κόμβου του B-δένδρου, ενώ ένας κόμβος B-δένδρου κατασκευάζεται λογικά, συλλέγοντας και αναλύοντας συντακτικά (parsing) όλες τι σχετιζόμενες με αυτόν μονάδες ευρετηρίου. Από τη στιγμή που το μέγεθος των μονάδων ευρετηρίου είναι σχετικά μικρό (συγκρινόμενο με το μέγεθος σελίδας), η υιοθέτησή τους εμποδίζει περιττά δεδομένα (redundant data) από το να γράφονται στη μνήμη φλας πολύ συχνά. Για την εξοικονόμηση αποθηκευτικού χώρου κατά την αποθήκευση των μονάδων ευρετηρίου, πολλές μονάδες ευρετηρίου που ανήκουν σε διαφορετικούς κόμβους, τοποθετούνται στους ίδιους τομείς. Αποτέλεσμα του παραπάνω, είναι το γεγονός ότι οι μονάδες ευρετηρίου ενός κόμβου μπορεί να είναι διασκορπισμένες σε διαφορετικούς τομείς μέσα στη μνήμη φλας.

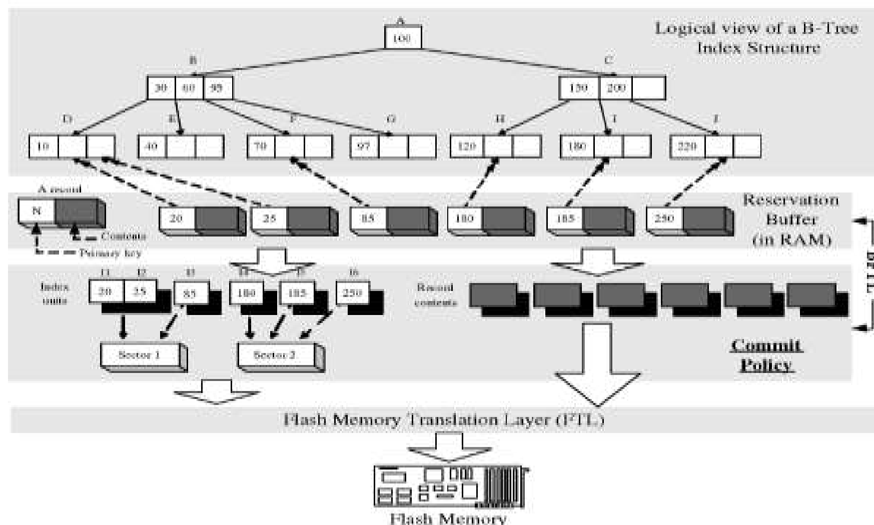
Μία μονάδα ευρετηρίου αποτελείται από πολλά συστατικά μέρη. Πιο συγκεκριμένα, περιέχει: ένα δείκτη προς τα δεδομένα (data_ptr), έναν κόμβο πατέρα (parent_node), έναν πρωτεύον κλειδί (primary_key), έναν αριστερό δείκτη (left_ptr), έναν δεξιό δείκτη (right_ptr), ένα αναγνωριστικό αριθμό (identifier number) καθώς και μία σημαία λειτουργίας (op_flag). Ο δείκτης προς τα δεδομένα, ο κόμβος πατέρα, ο αριστερός δείκτης, ο δεξιός δείκτης και το πρωτεύον κλειδί, είναι στοιχεία του αρχικού (original) κόμβου B-δένδρου. Αναπαριστούν, μία αναφορά προς το σώμα της καταχώρισης (record body), έναν δείκτη προς τον κόμβο πατέρα, έναν δείκτη προς το αριστερό κόμβο-παιδί, έναν δείκτη προς το δεξιό κόμβο-παιδί καθώς και το πρωτεύον κλειδί αντίστοιχα. Εκτός όμως από τα συστατικά του αρχικού κόμβου B-δένδρου, είναι απαραίτητος και ένας αναγνωριστικός αριθμός (identifier number). Ο αναγνωριστικός αριθμός μίας μονάδας ευρετηρίου, υποδηλώνει τον κόμβο B-δένδρου στον οποίο ανήκει η μονάδα ευρετηρίου. Η σημαία λειτουργίας (op_flag) καθορίζει τη λειτουργία που επιτελείται από τη μονάδα ευρετηρίου. Η λειτουργία μπορεί να είναι μία εισαγωγή (insertion), μία διαγραφή (deletion) ή μία ανανέωση (update). Επιπλέον, προστίθενται χρονοσφραγίδες (time-stamps) σε κάθε σύνολο μονάδων ευρετηρίου που προωθούνται στη μνήμη φλας, έτσι ώστε το BFTL να μη χρησιμοποιεί παλαιές (stale) μονάδες ευρετηρίου. Σημειώνουμε εδώ ότι το BFTL

χρησιμοποιεί το FTL για την αποθήκευση των μονάδων ευρετηρίου στη μνήμη φλας. Οι μονάδες ευρετηρίου όμως διασκορπίζονται στη μνήμη φλας. Η λογική πλευρά (logical view) ενός κόμβου Β-δένδρου, κατασκευάζεται με τη βοήθεια του BFTL. Παρ'όλα αυτά, η σάρωση της μνήμης φλας για τη συλλογή των μονάδων ευρετηρίου ενός κόμβου Β-δένδρου, είναι μία τρομερά μη αποδοτική διαδικασία. Γι'αυτό το λόγο,το BFTL χρησιμοποιεί έναν πίνακα μετάφρασης κόμβων που διαχειρίζεται τη συλλογή των μονάδων ευρετηρίου.

4.2.2.3 Πολιτική μεταφοράς (Commit Policy)

Ένα βασικό τεχνικό ζήτημα είναι η αποδοτική τοποθέτηση πολλών μονάδων ευρετηρίου μέσα σε λίγους τομείς (sectors). Σε αυτή την ενότητα, θα παρουσιάσουμε μία πολιτική μεταφοράς (commit policy) για τις μονάδες ευρετηρίου. Αρχικά, η πολιτική μεταφοράς ορίζει την ειδική περιοχή προσωρινής αποθήκευσης (reservation buffer) ως εξής: η ειδική περιοχή προσωρινής αποθήκευσης είναι μία περιοχή εγγραφής δεδομένων η οποία βρίσκεται στη μνήμη RAM. Όταν ένας κόμβος του Β-δένδρου, εισάγεται, διαγράφεται ή τροποποιείται, τότε κάθε νέα καταχώριση (record) που προκύπτει, αποθηκεύεται προσωρινά στην ειδική περιοχή προσωρινής αποθήκευσης (reservation buffer). Οι καταχωρίσεις της ειδικής περιοχή προσωρινής αποθήκευσης, αντιπροσωπεύουν λειτουργίες που δεν έχουν ακόμη εφαρμοστεί στο Β-δένδρο. Για κάθε καταχώριση r της ειδικής περιοχής προσωρινής αποθήκευσης, υπάρχει ένας αντίστοιχος κόμβος στο Β-δένδρο στον οποίο ανήκει η r.

Η τοποθέτηση των «βρώμικων» καταχωρίσεων (dirty records) στην ειδική περιοχή προσωρινής αποθήκευσης, προφυλάσει τις δομές ευρετηρίου Β-δένδρων της μνήμης φλας, από το να τροποποιούνται διαρκώς. Παρ'όλα αυτά, η χωρητικότητας της ειδικής περιοχής προσωρινής αποθήκευσης δεν είναι απεριόριστη. Πιο συγκεκριμένα, όταν η ειδική περιοχή προσωρινής αποθήκευσης γεμίσει, ορισμένες «βρώμικες» καταχωρίσεις μεταφέρονται στη μνήμη φλας. Στο σημείο αυτό, αξίζει να αναφέρουμε ότι οι δημιουργοί του BFTL προτείνουν τη μεταφορά στη μνήμη φλας ολόκληρου του περιεχομένου της ειδικής περιοχής προσωρινής αποθήκευσης. Η πολιτική μεταφοράς παρουσιάζεται παρακάτω με ένα παράδειγμα:



Σχήμα 4.3: Η πολιτική μεταφοράς (commit policy) τοποθετεί τις μονάδες ευρετηρίου σε τομείς της μνήμης φλας.

Ο χειρισμός ενός Β-δένδρου, όπως φαίνεται και στο σχήμα 4.3, χωρίζεται σε τρία μέρη: τη λογική πλευρά (logical view) του Β-δένδρου, το BFTL και το FTL. Υποθέτουμε ότι η ειδική περιοχή προσωρινής αποθήκευσης αποθηκεύει έξι καταχωρίσεις (records), των οποίων τα πρωτεύοντα κλειδιά είναι 20, 25, 85, 180, 185 και 250 αντίστοιχα. Όταν η ειδική περιοχή προσωρινής αποθήκευσης γεμίσει, οι καταχωρίσεις πρέπει να γραφτούν στη μνήμη φλας. Το BFTL παράγει πρώτα έξι μονάδες ευρετηρίου (I1-I6) για τις έξι καταχωρίσεις. Με βάση τα πρωτεύοντα κλειδιά των καταχωρίσεων και το εύρος τιμών των κόμβων φύλλου (D, E, F, G, H, I και J στο σχήμα 4.3), οι μονάδες ευρετηρίου χωρίζονται σε πέντε ξένα σύνολα (disjoint sets): $\{I1, I2\} \in D$, $\{I3\} \in F$, $\{I4\} \in H$, $\{I5\} \in I$ και $\{I6\} \in J$. Ο παραπάνω διαχωρισμός, εμποδίζει τις μονάδες ευρετηρίου του ίδιου κόμβου από το να διασκορπιστούν μεταξύ τους. Υποθέτουμε ότι οι τομείς (sectors), που παρέχονται από το FTL, αποθηκεύουν τρεις μονάδες ευρετηρίου ο καθένας. Έτσι, οι {I1, I2} και {I3} τοποθετούνται στον πρώτο τομέα (sector 1), ενώ οι {I4}, {I5} και {I6} τοποθετούνται στον δεύτερο τομέα (sector 2), μιάς και ο πρώτος τομέας έχει γεμίσει. Κατ'αυτόν τον τρόπο, πραγματοποιούνται μόνο δύο εγγραφές τομέων. Αν όμως δεν χρησιμοποιούνταν η ειδική περιοχή προσωρινής αποθήκευσης και η πολιτική μεταφοράς, τότε θα απαιτούνταν έξι εγγραφές τομέων για το χειρισμό των τροποποιήσεων της δομής ευρετηρίου.

Θεώρημα 4.1. *Το πρόβλημα της τοποθέτησης των μονάδων ευρετηρίου σε τομείς είναι NP-Hard.*

Απόδειξη. Θα αναγάγουμε το πρόβλημα της τοποθέτησης των μονάδων ευρετηρίου σε τομείς, στο πρόβλημα της **τοποθέτησης αντικειμένων σε κάδους (bin packing problem)**. Έστω B η χωρητικότητα του κάδου (bin) και K ο αριθμός των αντικειμένων (items). Κάθε αντικείμενο έχει ένα μέγεθος. Το πρόβλημα είναι να τοποθετήσουμε τα αντικείμενα σε όσο το δυνατόν μικρότερο αριθμό κάδων. Η αναγωγή γίνεται ως εξής: έστω ότι η χωρητικότητα B ενός κάδου είναι το μέγεθος ενός τομέα (sector) και κάθε αντικείμενο (item) είναι ένα ξένο σύνολο μονάδων ευρετηρίου. Ο αριθμός των ξένων συνόλων είναι ίδιος με τον αριθμό των αντικειμένων, δηλαδή K. Το μέγεθος ενός ξένου συνόλου είναι ίδιο με το μέγεθος του αντίστοιχου αντικειμένου. Εάν υπάρχει μία λύση στο πρόβλημα της τοποθέτησης των μονάδων ευρετηρίου σε τομείς, τότε η λύση είναι μοναδική και στο πρόβλημα της τοποθέτησης αντικειμένων σε κάδους (bin packing problem).

Algorithm 1. The FIRST-FIT-based Commit Policy

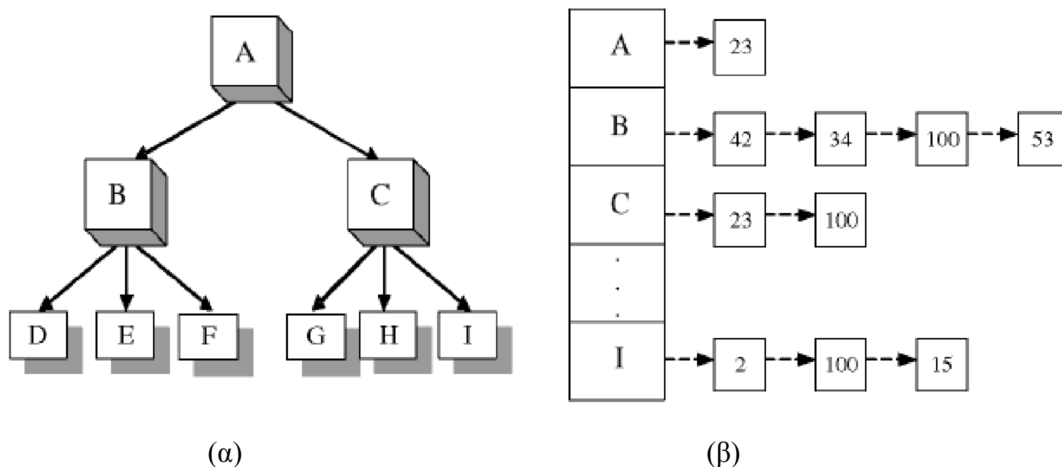
1: Let Φ denote the set of the disjoint sets of index units
2: Let Θ denote the set of the sectors
3: **while** Φ is not empty **do**
4: Let ds be a disjoint set in Φ
5: **if** there exists a used sector sec in Θ that has available free space for ds **then**
6: ds is stored in sector sec
7: **else**
8: create a new sector $nsec$ to store ds
9: $\Theta \leftarrow \Theta + nsec$
10: **end if**
11: $\Phi \leftarrow \Phi - ds$
12: **end while**
13: flush out Θ to flash memory

Υπάρχουν πολλοί προσεγγιστικοί αλγόριθμοι για το πρόβλημα της τοποθέτησης αντικειμένων σε κάδους (bin packing problem). Το BFTL υιοθετεί τον παραπάνω αλγόριθμο «πρώτου ταιριάσματος» (FIRST-FIT algorithm) για να περιορίσει (bound) τον πλήθος των σελίδων που γράφονται. Υποθέτουμε ότι ένας κόμβος B-δένδρου «χωράει» σε έναν τομέα, έτσι ώστε το μέγεθος ενός ξένου συνόλου να μην ξεπερνάει αυτό του τομέα.

4.2.2.4 Ο πίνακας μετάφρασης κόμβων (Node translation table)

Από τη στιγμή που η πολιτική μεταφοράς (commit policy) διασκορπίζει τις μονάδες ευρετηρίου ενός κόμβου του B-δένδρου στη μνήμη φλας, το BFTL χρησιμοποιεί τον **πίνακα μετάφρασης κόμβων (node translation table)**, με τη βοήθεια του οποίου η συλλογή των μονάδων ευρετηρίου ενός κόμβου είναι αποδοτική. Στην ενότητα αυτή θα παρουσιάσουμε τον τρόπο λειτουργίας του πίνακα μετάφρασης κόμβων.

Η κατασκευή της λογικής πλευράς (logical view) ενός κόμβου B-δένδρου, απαιτεί τη συλλογή όλων των σχετιζόμενων, με τον συγκεκριμένο κόμβο, μονάδων ευρετηρίου. Για αυτό το λόγο, η συλλογή των μονάδων ευρετηρίου ενός κόμβου πρέπει να γίνεται με αποδοτικό τρόπο. Ένας πίνακας μετάφρασης κόμβων χρησιμοποιείται σαν επικουρική δομή δεδομένων (auxiliary data structure), για την αποδοτική συλλογή των μονάδων ευρετηρίου. Ο πίνακας μετάφρασης κόμβων έχει αρκετές ομοιότητες με τον πίνακα μετάφρασης λογικών διευθύνσεων (logical address translation table), ο οποίος αντιστοιχίζει τη λογική διεύθυνση του μπλοκ (LBA-η διεύθυνση του τομέα) σε έναν φυσικό αριθμό σελίδας (physical page number). Ο πίνακας μετάφρασης κόμβων, αντιστοιχίζει έναν κόμβο B-δένδρου σε μία συλλογή από λογικές διευθύνσεων μπλοκ (LBA's), στις οποίες βρίσκονται αποθηκευμένες οι σχετιζόμενες με τον κόμβο μονάδες ευρετηρίου. Για τη δημιουργία της λογικής πλευράς ενός κόμβου B-δένδρου, το BFTL επισκέπτεται (διαβάζει) όλους τους τομείς στους οποίους βρίσκονται οι σχετιζόμενες με τον κόμβο μονάδες ευρετηρίου, και μετά κατασκευάζει μία ενημερωμένη λογική πλευρά (an up-to-date logical view) του κόμβου. Ο πίνακας μετάφρασης κόμβων, ανακατασκευάζεται σαρώνοντας τη μνήμη φλας, κάθε φορά που εκκινείται (powered-up) το σύστημα.



(α) (β)

Σχήμα 4.4: (α) Λογική πλευρά ενός B-δένδρου. (β) Πίνακας μετάφρασης κόμβων.

Το σχήμα 4.4 (α) παρουσιάζει ένα Β-δένδρο με εννέα κόμβους. Το σχήμα 4.4 (β) παρουσιάζει ένα πιθανό στιγμιότυπο του πίνακα μετάφρασης κόμβων, όπου κάθε κόμβος του Β-δένδρου αποτελείται από πολλές μονάδες ευρετηρίου, οι οποίες αποθηκεύονται σε διάφορους τομείς. Οι LBA's των τομέων ενώνονται μεταξύ τους σε μία λίστα, αμέσως μετά την αντίστοιχη εγγραφή (entry-η εγγραφή περιέχει το όνομα του κόμβου, όπως φαίνεται και στο σχήμα 4.4(β)) του πίνακα μετάφρασης κόμβων. Όταν γίνεται επίσκεψη ενός κόμβου του Β-δένδρου, όλες οι μονάδες ευρετηρίου που ανήκουν σε αυτόν συλλέγονται, σαρώνοντας όλους τους τομείς των οποίων οι LBA's αποθηκεύονται στη λίστα. Για παράδειγμα, για την κατασκευή της λογικής πλευράς του κόμβου C, στο σχήμα 4.4 (α), διαβάζονται από το BFTL (μέσω του FTL) οι λογικές διευθύνσεις μπλοκ (LBA) 23 και 100, για τη συλλογή των απαραίτητων μονάδων ευρετηρίου. Ένας τομέας, μπορεί να περιέχει μονάδες ευρετηρίου που ανήκουν σε διαφορετικούς κόμβους. Το σχήμα 4.4 (β) δείχνει ότι ο τομέας με λογική διεύθυνση μπλοκ (LBA) 100 περιέχει μονάδες ευρετηρίου των κόμβων B, C και I. Έτσι, όταν πραγματοποιείται εγγραφή σε ένα τομέα, η λογική διεύθυνση (LBA) του τομέα προστίθεται στις ανάλογες εγγραφές (entries) του πίνακα μετάφρασης κόμβων.

Παρ'όλα αυτά, οι λίστες του πίνακα μετάφρασης κόμβων μπορούν να μεγαλώσουν απροσδόκητα. Για παράδειγμα, εάν η λίστα μίας εγγραφής του πίνακα μετάφρασης κόμβων, αποτελείται από 100 στοιχεία (αποθηκεύει 100 διευθύνσεις τομέων), τότε η επίσκεψη του αντίστοιχου κόμβου απαιτεί 100 αναγνώσεις τομέων. Κάτι τέτοιο όμως, μπορεί να μειώσει σε μεγάλο βαθμό την απόδοση του συστήματος καθώς και να καταναλώσει μεγάλες ποσότητες μνήμης RAM. Για την αντιμετώπιση του παραπάνω προβλήματος, προτείνεται η **συμπίεση (compaction)** του πίνακα μετάφρασης κόμβων, όποτε είναι απαραίτητο. Μία παράμετρος συστήματος C, χρησιμοποιείται για να ελέγχει το μέγιστο μέγεθος των λιστών του πίνακα μετάφρασης κόμβων. Πιο συγκεκριμένα, όταν το μέγεθος μίας λίστας υπερβαίνει την παράμετρο C, τότε η λίστα συμπιέζεται. Για τη συμπίεση μίας λίστας, όλες οι σχετικές μονάδες ευρετηρίου συλλέγονται στη μνήμη RAM και στη συνέχεια γράφονται στη μνήμη φλας, σε όσο το δυνατόν μικρότερο αριθμό τομέων. Ως αποτέλεσμα των παραπάνω, το μέγεθος του πίνακα μετάφρασης κόμβων έχει ως ασυμπτωτικό άνω όριο το $O(N*C)$, όπου το N συμβολίζει το πλήθος των κόμβων του Β-δένδρου. Από την άλλη πλευρά, ο αριθμός αναγνώσεων σε επίπεδο τομέων, που απαιτούνται κατά την επίσκεψη ενός κόμβου, φράσεται από την παράμετρο C. Παρακάτω, παρουσιάζεται ένας αναθεωρημένος αλγόριθμος της πολιτικής μεταφοράς (commit policy) ο οποίος διαχειρίζεται τις λειτουργίες του πίνακα μετάφρασης κόμβων.

Algorithm 2. A Revised Commit Policy with the Considerations of the Node-Translation Table

```
1: Let Φ denote the set of the disjoint sets of index units
2: Let Θ denote the set of the sectors
3: Let ntt denote a node-translation table
4: while Φ is not empty do
5:    Let ds be a disjoint set in Φ
6:    Let en be the corresponding entry of ntt of a B-tree node that ds would update
7:    if the length of the list of the corresponding entry en is beyond C then
8:       execute the compaction of the list
9:    end if
10:   if there exists a used sector sec in Θ that has available free space for ds then
11:      ds is stored in sector sec
12:      record the LBA of sec in the list after the corresponding entry en of ntt
13:   else
14:      create a new sector nsec to store ds
15:      record the LBA of nsec in the list after the corresponding entry en of ntt
16:      Θ ← Θ + nsec
17:   end if
18:   Φ ← Φ - ds
19: end while
20: flush out Θ to flash memory
```

4.2.3 Ανάλυση πολυπλοκότητας του BFTL

Σε αυτή την ενότητα θα παρουσιάσουμε την ανάλυση της συμπεριφοράς του FTL και του BFTL. Ειδικότερα, θα υπολογίσουμε το πλήθος των τομέων που διαβάζονται και γράφονται από το FTL και το BFTL αντίστοιχα, για την εισαγωγή *n* καταχωρίσεων (records).

Υποθέτουμε ότι μία δομή ευρετηρίου B-δένδρου βρίσκεται στη μνήμη φλας, και ότι ένας κόμβος του B-δένδρου «χωράει» σε έναν τομέα (που παρέχεται από το FTL). Έστω ότι πρέπει να εισάγουμε *n* καταχωρίσεις. Οι τιμές όλων των πρωτεύοντων κλειδιών (primary keys) των *n* καταχωρίσεων, είναι διαφορετικές μεταξύ τους.

Αρχικά θα ερευνήσουμε τη συμπεριφορά του FTL. Ένας κόμβος B-δένδρου, αποθηκεύεται σε έναν ακριβώς τομέα στο FTL. Απαιτείται μία εγγραφή τομέα για κάθε εισαγωγή πρωτεύοντος κλειδιού, με την προυπόθεση βέβαια ότι δεν παρουσιάζεται υπερχείλιση κόμβου (node overflow). Αν συμβεί υπερχείλιση κόμβου, τότε ένα πρωτεύον κλειδί του κόμβου μεταφέρεται στον κόμβο πατέρα, και ο κόμβος διάσπαται (split) σε δύο νέους κόμβους. Από την άλλη πλευρά, εάν ένας κόμβος δεν είναι γεμάτος κατά το ήμισυ, τότε αυτός ο κόμβος μπορεί να συγχωνευτεί με άλλους μισογεμάτους κόμβους αδέρφια (sibling nodes) ή να εναλλάξει (rotate) ένα πρωτεύον κλειδί με έναν από τους κόμβους αδέρφια. Η διάσπαση κόμβου, απαιτεί τρεις εγγραφές τομέων υπό το FTL (δύο εγγραφές τομέων γίνονται για τους δύο νέους κόμβους που δημιουργούνται, και μία εγγραφή τομέα για τον κόμβο πατέρα). Τόσο η συγχώνευση (merging) όσο και η εναλλαγή (rotating), απαιτούν έκαστως, το πολύ τρεις εγγραφές τομέων υπό το FTL. Έστω H το τρέχον ύψος του B-δένδρου, N_{split} ο αριθμός των κόμβων που υφίστανται διάσπαση και $N_{merge/rotate}$ ο αριθμός των κόμβων που συγχωνεύονται/εναλλάσσονται, κατά τη διάρκεια εισαγωγής καταχωρίσεων. Οι αριθμοί των τομέων που διαβάζονται και γράφονται από το FTL για το χειρισμό των εισαγωγών, φαίνονται παρακάτω:

$$\begin{cases} R_{FTL} = O(n * H) \\ W_{FTL} = O(n + 3 * N_{split} + 3 * N_{merge/rotate}) \end{cases} \quad (4.1)$$

Υποθέτουμε τώρα ότι το μέγεθος του τομέα παραμένει ίδιο και υπό το BFTL (υπενθυμίζουμε εδώ ότι το BFTL είναι πάνω από το FTL), και ότι το ύψος του B-δένδρου είναι *H*. Ας υπολογίσουμε τον αριθμό των τομέων που διαβάζονται και γράφονται στη μνήμη φλας, υπό το BFTL, όταν εισάγονται *n* καταχωρίσεις. Επειδή το BFTL υιοθετεί τον πίνακα μετάφρασης κόμβων για τη συλλογή των μονάδων ευρετηρίου (index units) ενός κόμβου, το πλήθος των τομέων που διαβάζονται για την κατασκευή ενός κόμβου B-δένδρου, εξαρτάται από το μέγεθος των λιστών του πίνακα μετάφρασης κόμβων. Έστω ότι το μέγεθος της λίστας έχει ως άνω όριο το *C* (που αναφέραμε στην ενότητα 4.2.2.4). Τότε, ο αριθμός των τομέων που διαβάζονται από το BFTL για το χειρισμό των εισαγωγών, παρουσιάζεται παρακάτω:

$$R_{BFTL} = O(n * H * C) \quad (4.2)$$

Η σχέση (4.2) δείχνει ότι το BFTL διαβάζει περισσότερους τομείς από το FTL για το χειρισμό των εισαγωγών. Στην πραγματικότητα, το BFTL ανταλλάσσει αποδοτικά εγγραφές με επιπλέον (extra) αναγνώσεις. Ο αριθμός των τομέων που εγγράφονται από το BFTL, υπολογίζεται ως εξής: Επειδή το BFTL υιοθετεί την ειδική περιοχή προσωρινής αποθήκευσης (reservation buffer) για να διατηρεί τις καταχωρίσεις (records) στη RAM, τις οποίες στη συνέχεια μεταφέρει στη μνήμη φλας, οποιεσδήποτε τροποποιήσεις σε κόμβους του B-δένδρου (δηλ. οι μονάδες ευρετηρίου) τοποθετούνται (packed) σε λίγους μόνο τομείς. Έστω ότι η χωρητικότητα της ειδικής περιοχή προσωρινής αποθήκευσης είναι *b* καταχωρίσεις. Κατ'αυτόν τον τρόπο, το περιεχόμενο της ειδικής περιοχής προσωρινής αποθήκευσης, μεταφέρεται στη μνήμη φλας από την πολιτική μεταφοράς (commit policy), τουλάχιστον $\lceil n/b \rceil$ φορές κατά τη διάρκεια του χειρισμού των *n* εισαγωγών. Έστω N^i_{split} και $N^i_{merge/rotate}$ οι αριθμοί των κόμβων που διασπόνται και συγχωνεύονται/εναλλάσσονται, αντίστοιχα, κατά το χειρισμό της i-οστής μεταφοράς του περιεχομένου της ειδικής περιοχής προσωρινής αποθήκευσης στη μνήμη φλας. Είναι φανερό ότι, $\sum_{i=1}^{\lceil n/b \rceil} N^i_{split} = N_{split}$ και $\sum_{i=1}^{\lceil n/b \rceil} N^i_{merge/rotate} = N_{merge/rotate}$ μιάς και τα B-δένδρα υπό το FTL και το BFTL είναι λογικά πανομοιότυπα (logically identical).

Σε κάθε ξεχωριστό βήμα της μεταφοράς του περιεχομένου της ειδικής περιοχής προσωρινής αποθήκευσης στη μνήμη φλας, μεταφέρονται $(b + N^i_{split} * fanout + N^i_{merge/rotate} * fanout)$ «βρώμικες» μονάδες ευρετηρίου (dirty index units). Υπενθυμίζουμε εδώ ότι το fanout είναι ο μέγιστος αριθμός παιδιών ανά κόμβο. Ο όρος $N^i_{split} * fanout$ υπάρχει στον παραπάνω τύπο, επειδή κάθε διάσπαση κόμβου έχει ως αποτέλεσμα την ανανέωση του κόμβου πατέρα και δύο νέων κόμβων όπου χρειάζονται fanout σε πλήθος, μονάδες ευρετηρίου. Ο όρος $N^i_{merge/rotate} * fanout$ υποδηλώνει ότι κάθε συγχώνευση/εναλλαγή απαιτεί το πολύ fanout σε πλήθος, μονάδες σβησίματος (δηλ. κάθε συγχώνευση έχει ως αποτέλεσμα τη δημιουργία ενός νέου κόμβου και την ανανέωση του κόμβου πατέρα, στην οποία απαιτούνται fanout σε πλήθος μονάδες ευρετηρίου). Υποθέτουμε όπως και στο FTL, ότι ένας κόμβος B-

δένδρου «χωράει» σε έναν τομέα. Αυτό σημαίνει ότι ένας τομέας αποθηκεύει (fanout-1) μονάδες ευρετηρίου. Έστω Λ=(fanout-1). Ο αριθμός των τομέων που γράφονται, κατά την i-οστή μεταφορά του περιεχομένου της ειδικής περιοχής προσωρινής αποθήκευσης στη μνήμη φλας, είναι περίπου ίσος με $(\frac{b}{\Lambda} + N^i_{split} + N^i_{merge/rotate})$. Για την ολοκληρωτική μεταφορά του περιεχομένου της ειδικής περιοχής προσωρινής αποθήκευσης στη μνήμη φλας, πρέπει να γραφτούν τουλάχιστον $\sum_{i=1}^{\lceil n/b \rceil}(\frac{b}{\Lambda} + N^i_{split} + N^i_{merge/rotate}) = (\sum_{i=1}^{\lceil n/b \rceil}\frac{b}{\Lambda}) + N_{split} + N_{merge/rotate}$ τομείς.

Από τη στιγμή που το BFTL υιοθετεί τον αλγόριθμο «πρώτου ταιριάσματος» (FIRST-FIT algorithm), ο αριθμός των τομέων που γράφονται από το BFTL δίνεται από τον παρακάτω τύπο:

$$W_{BFTL} = O\left(2 * \left(\sum_{i=1}^{\lceil n/b \rceil}\frac{b}{\Lambda} + N_{split} + N_{merge/rotate}\right)\right) = O\left(\frac{2*n}{\Lambda} + 2*N_{split} + 2*N_{merge/rotate}\right) \quad (4.3)$$

Συκγρίνοντας τις σχέσεις (4.1) και (4.3), παρατηρούμε ότι το W_{BFTL} είναι πολύ μικρότερο από το W_{FTL} , από τη στιγμή που το Λ (ο αριθμός των μονάδων ευρετηρίου που αποθηκεύονται σε έναν τομέα) είναι μεγαλύτερο από δύο. Παρ'όλα αυτά, η συμπίεση (compaction) του πίνακα μετάφρασης κόμβων, που παρουσιάστηκε στην ενότητα 4.2.2.4, εισάγει επιπλέον φόρτο κατά τα χρόνο εκτέλεσης. Παρακάτω θα δείξουμε ότι, όταν το Λ ισούται με 20, ο αριθμός των τομέων που γράφονται από το BFTL είναι μεταξύ 1/5 και 1/26 του αριθμού των τομέων που γράφονται από το FTL.

4.2.4 Εκτίμηση της απόδοσης του BFTL

Στην ενότητα αυτή εκτιμάται η απόδοση του BFTL με σκοπό να παρουσιαστούν τα οφέλη από την υιοθέτησή του. Περιορίζοντας τις περιττές εγγραφές δεδομένων στη μνήμη φλας, οι επιδόσεις των λειτουργιών/πράξεων πάνω στα Β-δένδρα βελτιώνονται αισθητά.

4.2.4.1 Επισκόπηση πειραμάτων

Ένα σύστημα βασισμένο σε NAND μνήμη φλας, χρησιμοποιήθηκε για την εκτίμηση της απόδοσης του BFTL. Το μέγεθος της NAND μνήμης φλας είναι 4MB. Για να εκτιμηθεί η απόδοση του FTL, ένα Β-δένδρο κατασκευάστηκε απευθείας από το FTL. Η άπληστη πολιτική ανακύκλωσης μπλοκ (greedy block-recycling policy), υιοθετήθηκε από το FTL για τη συλλογή απορριμμάτων.

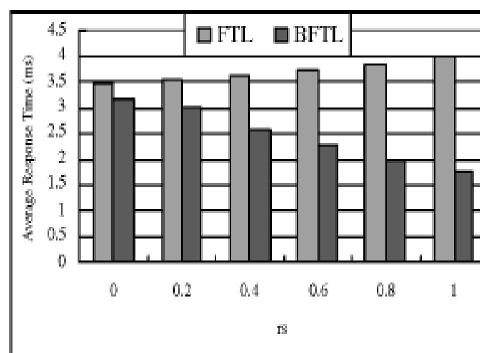
Μεγάλη βαρύτητα δόθηκε στην εκτίμηση της απόδοσης των λειτουργιών ευρετηρίου (index operations). Ο αριθμός fanout που χρησιμοποιήθηκε στα πειράματα είναι ίσος με 21 και το μέγεθος ενός κόμβου Β-δένδρου «χωράει» σε έναν τομέα. Για την εκτίμηση της απόδοσής του, το BFTL διαμορφώθηκε ως εξής: η ειδική περιοχή προσωρινής αποθήκευσης (reservation buffer) αποθηκεύει 60

καταχωρίσεις, ενώ οι λίστες του πίνακα μετάφρασης κόμβων (node translation table) έχουν ως μέγιστο μέγεθος τα τέσσερα στοιχεία.

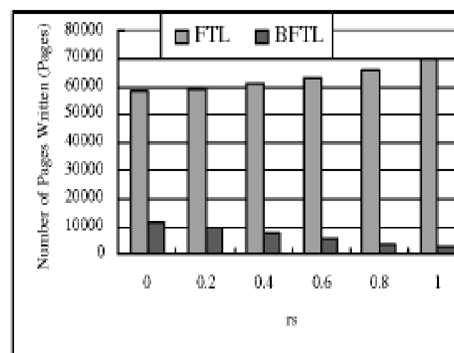
Στα πειράματα που πραγματοποιήθηκαν, μετρήθηκαν οι μέσοι χρόνοι απόκρισης εισαγωγών (insertions) και διαγραφών (deletions). Όσο μικρότερος είναι ο μέσος χρόνος απόκρισης, τόσο πιο αποδοτική είναι η λειτουργία εισαγωγής/διαγραφής. Επίσης, μετρήθηκε το πλήθος των σελίδων που αναγνώστηκαν (page read), το πλήθος σελίδων που γράφτηκαν (page write) καθώς και το πλήθος των σβησμένων μπλοκ (erased blocks). Ακόμη, μετρήθηκε η κατανάλωση ενέργειας του BFTL και του FTL. Τα αποτελέσματα των πειραμάτων παρουσιάζονται στις ενότητες που ακολουθούν.

4.2.4.2 Απόδοση δημιουργίας Β-δένδρων

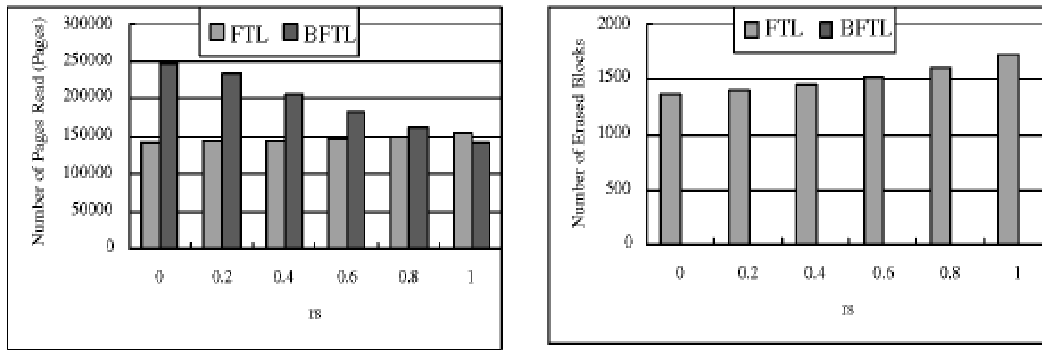
Στην ενότητα αυτή, παρουσιάζουμε τα αποτελέσματα των πειραμάτων σχετικά με την απόδοση του FTL και του BFTL, κατά τη δημιουργία Β-δένδρων. Κατασκευάζονται δομές δεδομένων Β-δένδρου, με την εισαγωγή καταχωρίσεων (records). Σε κάθε νέα εκτέλεση των πειραμάτων, εισάγονται 30000 καταχωρίσεις. Παρόλο που ένα Β-δένδρο, που κατασκευάστηκε από 30000 καταχωρίσεις υπό το FTL, κατέλαβε 1197KB μνήμης φλας, το συνολικό ποσό δεδομένων που γράφτηκε από το FTL ήταν 14MB. Επειδή στα πειράματα χρησιμοποιήθηκε NAND μνήμη φλας μεγέθους 4MB, οι μηχανισμοί συλλογής απορριμμάτων ενεργοποιήθηκαν πολύ νωρίς για την ανάκτηση ελεύθερου χώρου. Στα πειράματα, ένας λόγος *rs* χρησιμοποιήθηκε για τον έλεγχο της κατανομής τιμών των εισαγόμενων κλειδιών: εάν το *rs* ισούται με το 0, όλα τα κλειδιά παράγονται τυχαία. Εάν το *rs* ισούται με 1, τότε οι τιμές των εισαγόμενων κλειδιών ακολουθούν αύξουσα σειρά. Τέλος, όταν το *rs* ισούται με 0.5, οι τιμές των μισών κλειδιών ακολουθούν αύξουσα σειρά ενώ τα υπόλοιπα κλειδιά παράγονται τυχαία. Στα σχήματα 4.5 και 4.6, ο άξονας των x συμβολίζει τις τιμές του *rs*.



(α) Μέσοι χρόνοι απόκρισης κατά την εισαγωγή 30000 καταχωρίσεων.



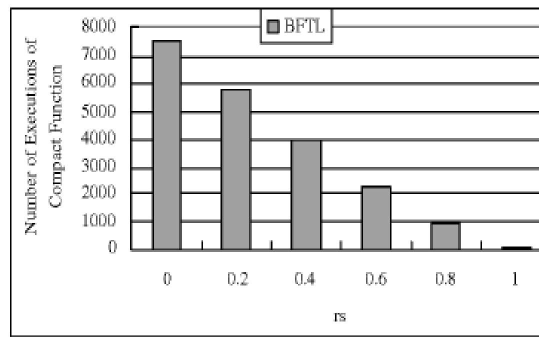
(β) Αριθμός σελίδων που γράφονται μετά την εισαγωγή 30000 καταχωρίσεων.



(γ) Αριθμός σελίδων που διαβάζονται μετά την εισαγωγή 30000 καταχωρίσεων.

(δ) Αριθμός των σβησμένων μπλοκ μετά την εισαγωγή 30000 καταχωρίσεων.

Σχήμα 4.5: (α), (β), (γ), (δ). Αποτελέσματα πειραμάτων κατά την δημιουργία δομών Β-δένδρου.



Σχήμα 4.6: Πλήθος εμφανίσεων λειτουργίας συμπίεσης του πίνακα μετάφρασης κόμβων.

Το σχήμα 4.5 (α) δείχνει το μέσο χρόνο απόκρισης εισαγωγών. Παρατηρούμε ότι το BFTL είναι πολύ πιο αποδοτικό από το FTL. Πιο συγκεκριμένα, ο χρόνος απόκρισης του BFTL είναι το 1/3 του αντίστοιχου χρόνου του FTL, όταν οι τιμές των κλειδιών ακολουθούν αύξουσα σειρά (*rs*=1). Το BFTL ξεπερνά σε απόδοση το FTL ακόμα και όταν οι τιμές των κλειδιών παράγονται τυχαία (*rs*=0). Όταν τα κλειδιά παράγονται ακολουθιακά (*rs*=1), ο αριθμός των τομέων που γράφονται μειώνεται, επειδή οι μονάδες ευρετηρίου που ανήκουν σε ένα κόμβο, δεν διασκορπίζονται σε μεγάλο βαθμό στη μνήμη φλας. Επιπρόσθετα, το μέγεθος των λιστών του πίνακα μετάφρασης κόμβων είναι σχετικά μικρό και η συμπίεση των λιστών δεν επιφέρει ιδιαίτερο φόρτο.

Όπως είναι γνωστό, η εγγραφή στη μνήμη φλας είναι μία ιδιαίτερα «ακριβή» λειτουργία/πράξη, διότι φθείρει (wear) τη μνήμη φλας, καταναλώνει αρκετή ενέργεια και προκαλεί την εμφάνιση της διαδικασίας συλλογής απορριμμάτων. Τα σχήματα 4.5 (β) και 4.5 (γ) δείχνουν τον αριθμό των σελίδων που γράφονται και διαβάζονται αντίστοιχα, κατά την εκτέλεση των πειραμάτων. Οι αριθμοί αντανακλούν τη χρήση της μνήμης φλας από το FTL και το BFTL αντίστοιχα. Από τα σχήματα φαίνεται ότι, το BFTL ανταλλάσσει αποδοτικά εγγραφές με επιπλέον (extra) αναγνώσεις, υιοθετόντας την πολιτική μεταφοράς (commit policy). Από την άλλη πλευρά, οι επιπλέον αναγνώσεις προέρχονται από την επίσκεψη των τομέων για την κατασκευή

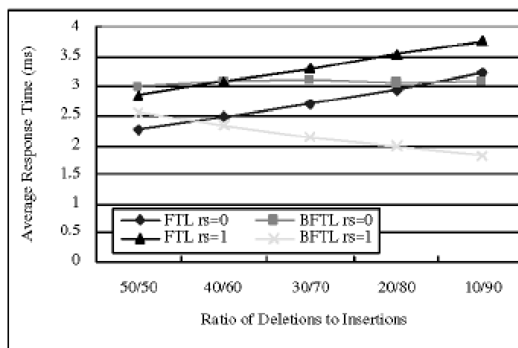
της λογικής πλευράς ενός κόμβου B-δένδρου, όπως αναφέραμε και στην ενότητα 4.2.2.4.

Στο σχήμα 4.5 (δ) φαίνεται ότι το BFTL υπερτερεί ολοκληρωτικά εις βάρος του FTL, όσον αφορά το μηχανισμό συλλογής απορριμμάτων. Πιο συγκεκριμένα, σε όλα τα πειράματα που πραγματοποιήθηκαν, ο μηχανισμός συλλογής απορριμμάτων δεν ενεργοποιήθηκε καθόλου. Ως αποτέλεσμα, το BFTL υπόσχεται μεγαλύτερους χρόνους ζωής (lifetime) στη μνήμη φλας.

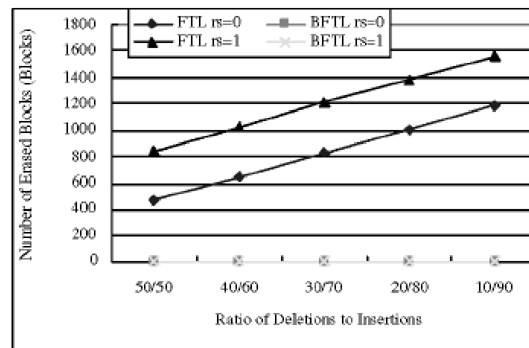
Το σχήμα 4.6 παρουσιάζει το φόρτο που εισάγεται στο σύστημα εξαιτίας της συμπίεσης (compaction) του πίνακα μετάφρασης κόμβων. Πιο συγκεκριμένα, το πλήθος εμφάνισης της λειτουργίας συμπίεσης, μειώνεται όταν οι τιμές των εισαγόμενων κλειδιών ακολουθούν αύξουσα σειρά. Από την άλλη πλευρά, το BFTL συμπιέζει συχνότερα τον πίνακα μετάφρασης κόμβων, όταν οι τιμές των εισαγόμενων κλειδιών παράγονται τυχαία, μιάς και οι μονάδες ευρετηρίου ενός κόμβου διασκορπίζονται τυχαία στους διάφορους τομείς. Έτσι, το μέγεθος των λιστών μεγαλώνει ταχύτατα με αποτέλεσμα να συμπιέζονται συχνότερα.

4.2.4.3 Απόδοση διατήρησης B-δένδρων

Στην ενότητα αυτή, παρουσιάζουμε τα αποτελέσματα των πειραμάτων σχετικά με την απόδοση του FTL και του BFTL, κατά τη διατήρηση των B-δένδρων. Πιο συγκεκριμένα, κατά τη διάρκεια των πειραμάτων, πραγματοποιήθηκαν εισαγωγές, διαγραφές και τροποποιήσεις καταχωρίσεων. Για παράδειγμα, το 30% των συνολικών πράξεων ήταν διαγραφές (deletions) και το υπόλοιπο 70% εισαγωγές (insertions). Σε κάθε πείραμα που εκτελέστηκε, πραγματοποιούνταν 30000 πράξεις πάνω στα B-δένδρα, με το λόγο διαγραφές/εισαγωγές να κυμαίνεται μεταξύ 50/50, 40/60, 30/70, 20/80 και 10/90. Η τιμή του *rs* ήταν 1 ή 0.



(α) Μέσοι χρόνοι απόκρισης κάτω από διάφορους λόγους διαγραφών/εισαγωγών.



(β) Αριθμός σβησμένων μπλοκ κάτω από διάφορους λόγους διαγραφών/εισαγωγών.

Σχήμα 4.7: (α), (β).Αποτελέσματα πειραμάτων κατά τη διατήρηση B-δένδρων.

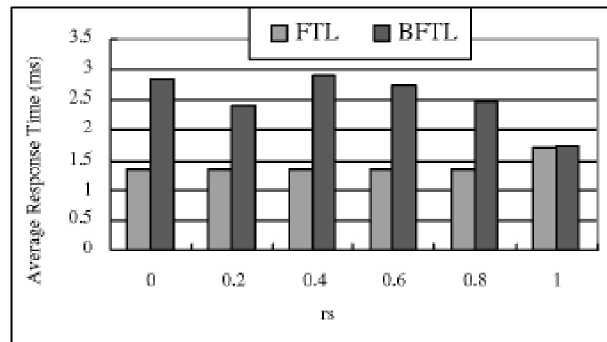
Το σχήμα 4.7 (α) δείχνει το μέσο χρόνο απόκρισης κάτω από διάφορους λόγους διαγραφών/εισαγωγών. Πιο συγκεκριμένα, οι μέσοι χρόνοι απόκρισης δείχνουν ότι το FTL είναι πιο αποδοτικό από το BFTL, όταν ο λόγος διαγραφές/εισαγωγές αλλάζει από 50/50 σε 20/80, με *rs*=0 (τα κλειδιά παράγονται τυχαία). Από τη στιγμή που τα εισαγόμενα κλειδιά παράγονται τυχαία, και στη συνέχεια διαγράφονται τυχαία, όλο

και περισσότερες μονάδες ευρετηρίου ενώνονται μεταξύ τους στον πίνακα μετάφρασης κόμβων, με αποτέλεσμα η επίσκεψη ενός κόμβου να μην είναι αποδοτική. Όταν ο λόγος διαγραφές/εισαγωγές είναι πάνω από το 20/80 με *rs*=0, το BFTL είναι πιο αποδοτικό από το FTL, επειδή ο αριθμός των εισαγωγών αυξήθηκε με αποτέλεσμα το FTL να γράψει περισσότερες σελίδες. Όταν ο λόγος διαγραφές/εισαγωγές αλλάζει από 50/50 σε 10/90 με *rs*=1, η απόδοση του BFTL βελτιώνεται ριζικά. Ο λόγος είναι ότι το BFTL γράφει λιγότερες σελίδες, όταν οι τιμές των εισαγόμενων κλειδιών ακολουθούν αύξουσα σειρά. Αυτό έχει ως αποτέλεσμα, το BFTL να έχει καλύτερη απόδοση από το FTL, όταν ο λόγος διαγραφές/εισαγωγές αλλάζει από 50/50 σε 10/90 με *rs*=1.

Το σχήμα 4.7 (β) δείχνει τον αριθμό των σβησμένων μπλοκ στο πείραμα. Οι δραστηριότητες συλλογής απορριμμάτων έχεουν μειωθεί σημαντικά στο BFTL. Ειδικότερα, δεν έχουν ακόμη ξεκινήσει, σε όλη τη διάρκεια των πειραμάτων.

4.2.4.4 Απόδοση αναζήτησης σε B-δένδρα

Στην ενότητα αυτή, παρουσιάζουμε τα αποτελέσματα των πειραμάτων σχετικά με την απόδοση του FTL και του BFTL, κατά την αναζήτηση στοιχείων σε B-δένδρα. Σε κάθε πείραμα που έλαβε χώρα, πραγματοποιήθηκαν 3000 αναζητήσεις κλειδιών με διαφορετικές τιμές, ενώ μετρήθηκαν οι μέσοι χρόνοι απόκρισης των αναζητήσεων. Το σχήμα 4.8 που ακολουθεί, παρουσιάζει τα αποτελέσματα των πειραμάτων.



Σχήμα 4.8: Μέσοι χρόνοι απόκρισης αναζητήσεων κλειδιών.

Από τη στιγμή που οι μονάδες ευρετηρίου των κόμβων του B-δένδρου διασκορπίζονται στη μνήμη φλας εξαιτίας της πολιτικής μεταφοράς (commit policy), το BFTL χρειάζεται να διαβάσει περισσότερους τομείς για την κατασκευή των κόμβων. Αυτό έχει ως αποτέλεσμα, στο BFTL οι λειτουργίες αναζήτησης στοιχείων να διαρκούν περισσότερο χρόνο απ’ότι στο FTL. Για την αποφυγή μεγάλων χρόνων αναζήτησης στο BFTL, πρέπει το μέγεθος των λιστών του πίνακα μετάφρασης κόμβων να έχει ένα άνω όριο *C*, όπως αναφέραμε και στην ενότητα 4.2.2.4. Στα πειράματα που πραγματοποιήθηκαν, το *C* έχει τιμή τέσσερα, και όπως φαίνεται και στο σχήμα 4.8, οι μέσοι χρόνοι απόκρισης των αναζητήσεων στο BFTL είναι το πολύ διπλάσιοι αυτών του FTL. Από την άλλη πλευρά, το BFTL μείωσε σημαντικά τον αριθμό των εγγραφών σε επίπεδο σελίδας καθώς και τον αριθμό των σβησμένων μπλοκ.

4.2.4.5 Κατανάλωση ενέργειας

Η κατανάλωση ενέργειας είναι ένας πάρα πολύ σημαντικός παράγοντας σε φορητές συσκευές (portable devices). Κατά τη διάρκεια των αναγνώσεων/ εγγραφών/ σβησιμάτων των μπλοκ (block erases) που πραγματοποιήθηκαν, μετρήθηκε η κατανάλωση ενέργειας του BFTL και του FTL αντίστοιχα. Η κατανάλωση ενέργειας κατά τη διάρκεια των αναγνώσεων/ εγγραφών/ σβησιμάτων μπλοκ μίας τυπικής NAND μνήμης φλας, παρουσιάζονται στον πίνακα 4.1. Η κατανάλωση ενέργειας κατά τη διάρκεια των πειραμάτων, παρουσιάζεται στον πίνακα 4.2 που ακολουθεί.

Creation		
	BFTL	FTL
$rs = 0$	27.09	28.33
$rs = 1$	14.79	32.65

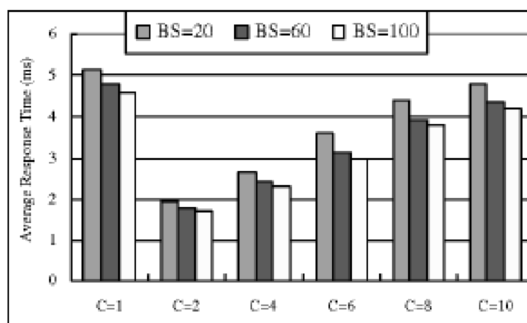
Maintenance		
	BFTL	FTL
50/50, $rs = 0$	25.28	18.52
50/50, $rs = 1$	21.56	23.24
10/90, $rs = 0$	26.15	26.25
10/90, $rs = 1$	15.53	30.47

Πίνακας 4.2: Κατανάλωση ενέργειας BFTL και FTL (σε joule).

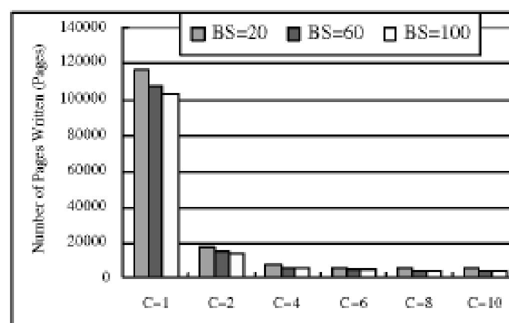
Η κατανάλωση ενέργειας του BFTL είναι φανερά μικρότερη από αυτή του FTL. Από τη στιγμή που οι εγγραφές σε επίπεδο σελίδας (page writes) και τα σβησίματα των μπλοκ (block erases), καταναλώνουν περισσότερη ενέργεια από τις αναγνώσεις σελίδων (page reads), το BFTL καταναλώνει λιγότερη ενέργεια από τη στιγμή που ανταλλάσσει αποδοτικά εγγραφές με επιπλέον (extra) αναγνώσεις. Επιπρόσθετα, η ενέργεια που καταναλώνεται εξαιτίας του μηχανισμού συλλογής απορριμμάτων, μειώθηκε από το BFTL, μιας και καταναλώνει τον ελεύθερο χώρο (free space) με μικρότερο ρυθμό από ότι το FTL.

4.2.4.6 Απόδοση BFTL για διάφορες τιμές του *C*

Στην ενότητα αυτή, εκτιμάται η απόδοση του BFTL κάτω από διάφορες τιμές της παραμέτρου *C*. Κατά τη διάρκεια των πειραμάτων, εισήχθησαν 30000 καταχωρίσεις, με *rs*=0.5 και *BS* μεταξύ 20, 60 και 100.



(α) Μέσοι χρόνοι απόκρισης για διάφορες τιμές του *C*.



(β) Αριθμός σελίδων που γράφονται κάτω από διάφορες τιμές του *C*.

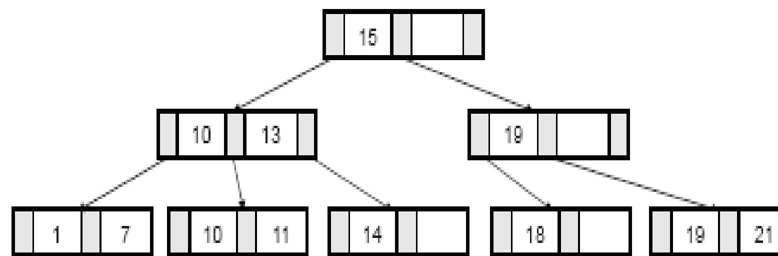
Σχήμα 4.9: (α), (β). Μέσοι χρόνοι απόκρισης και αριθμός σελίδων που γράφονατι, υπό διάφορες τιμές του *C*.

Όπως φαίνεται και στο παραπάνω σχήμα, όταν το *C* έχει τιμή 1, ο μέσος χρόνος απόκρισης είναι πολύ μεγάλος, επειδή πραγματοποιούνται πολλές συμπιέσεις στις λίστες του πίνακα μετάφρασης κόμβων. Όταν το *C* λαμβάνει τιμή 2, ο μέσος χρόνος απόκρισης είναι ο μικρότερος δυνατός. Παρ'όλα αυτά, σύμφωνα με το σχήμα 4.9 (β), όταν το *C* λαμβάνει τιμή ίση με 4, ο αριθμός των σελίδων που γράφονται μειώνονται σημαντικά σε σχέση με *C*=2. Επιπρόσθετα, παρατηρούμε ότι δεν παρουσιάζεται επιπλέον βελτίωση αν αυξήσουμε την τιμή του *C* πάνω από 4.

4.3 B$^+$- δένδρα

Τα **B$^+$- δένδρα** είναι μία δημοφιλής δομή δεδομένων για την αποδοτική διαχείριση μεγάλου όγκου δεδομένων. Διατηρούν τα δεδομένα ταξινομημένα ενώ παρέχουν λογαριθμικούς χρόνους ανάκτησης (retrieval), εισαγωγής (insertion) και διαγραφής (deletion) δεδομένων. Τα B$^+$- δένδρα, χρησιμοποιούνται ευρέως σε συστήμτα αρχείων καθώς και συστήματα διαχείρισης βάσεων δεδομένων (DBMSs).

Η δομή του B$^+$- δένδρου είναι παρόμοια με αυτή των δυαδικών δένδρων αναζήτησης (binary search trees). Καθώς κάθε κόμβος σε ένα δυαδικό δένδρο αναζήτησης περιέχει μόνο ένα κλειδί και δύο δείκτες, ένας κόμβος B$^+$- δένδρου περιλαμβάνει μέχρι *d* τιμές κλειδιών, $K_1, K_2, ..., K_d$, καθώς και *d+1* δείκτες, $P_1, P_2, ..., P_{d+1}$. Οι τιμές των κλειδιών που αποθηκεύονται σε ένα κόμβο, είναι ταξινομημένες. Ο μέγιστος αριθμός κλειδιών *d*, που αποθηκεύεται σε έναν κόμβο, ονομάζεται **τάξη (order)** του B$^+$- δένδρου. Το σχήμα 4.10 παρουσιάζει ένα B$^+$- δένδρο τάξης 2.



Σχήμα 4.10: B$^+$ - δένδρο τάξης 2.

Υπάρχουν δύο κατηγορίες κόμβων σε ένα B$^+$- δένδρο: οι κόμβοι φύλλα (leaf-nodes), και οι εσωτερικοί κόμβοι (interior nodes). Ένας κόμβος φύλλο, μπορεί να αποθηκεύσει από *d/2* μέχρι *d* κλειδιά μαζί με τους αντίστοιχους δείκτες. Για *i=1, 2,..., D* κάθε δείκτης P_i δείχνει προς την καταχώριση που σχετίζεται με το K_i. Στην περίπτωση των εσωτερικών κόμβων, οι οποίοι ονομάζονται και κόμβοι ευρετηρίου (index nodes), η δομή είναι βασικά ίδια με αυτή των κόμβων φύλλων, με τη διαφορά ότι οι δείκτες δείχνουν προς άλλους κόμβους του B$^+$- δένδρου. Το B$^+$- δένδρο,

προσπαθεί να διατηρήσει τους κόμβους του ισοζυγισμένους (balanced), οποτεδήποτε πραγματοποιείται μία πράξη εισαγωγής ή διαγραφής.

Για την εύρεση μίας καταχώρισης (record) με τιμή κλειδιού *K* σε ένα B^+- δένδρο, πραγματοποιείται επίσκεψη πολλών κόμβων, σε ένα μονοπάτι από τη ρίζα (root) προς τα φύλλα του δένδρου. Σε κάθε κόμβο, η διαδικασία ανάκτησης (retrieval process) αναζητά το μικρότερο κλειδί K_i που είναι μεγαλύτερο του *K*, και στη συνέχεια φορτώνει τον κόμβο που υποδεικνύεται από τον αντίστοιχο δείκτη P_i. Εάν δεν υπάρχει τέτοιο κλειδί, ακολουθούμε τον τελευταίο δείκτη P_m, όπου με *m* συμβολίζουμε τον αριθμό των δεικτών στο κόμβο. Όταν η διαδικασία ανάκτησης φτάνει σε έναν κόμβο φύλλο, ελέγχεται το κατά πόσο ο κόμβος περιέχει το κλειδί K_i το οποίο ισούται με *K*. Εάν ο κόμβος περιέχει ένα τέτοιο κλειδί K_i, τότε επιστρέφεται η καταχώριση στην οποία δείχνει ο δείκτης P_i. Σε διαφορετική περίπτωση, η διαδικασία ανάκτησης αποτυγχάνει.

Για την εισαγωγή ενός κλειδιού με τιμή *K*, η διαδικασία εισαγωγής (insertion process) βρίσκει πρώτα ένα κατάλληλο κόμβο φύλλο, χρησιμοποιώντας την παραπάνω διαδικασία ανάκτησης. Στη συνέχεια, εισάγει το κλειδί *K* στον κόμβο φύλλο. Εάν παρ'όλα αυτά, ο κόμβος φύλλο είναι γεμάτος, τότε λαμβάνει χώρα μία διάσπαση (split). Γενικά, για τη διαχείριση *d+1* κλειδιών, τα πρώτα $\lceil (d+1)/2 \rceil$ κλειδιά τοποθετούνται στον υπάρχοντα κόμβο και τα υπόλοιπα σε έναν καινούργιο κόμβο. Μετά τη διάσπαση, το κλειδί με τη μικρότερη τιμή του νέου κόμβου, εισάγεται στον κόμβο πατέρα του.

Στις περισσότερες περιπτώσεις, ο κόμβος πατέρας δεν είναι γεμάτος και η διαδικασία εισαγωγής τερματίζεται. Εάν όμως, ο κόμβος πατέρας είναι γεμάτος, η διάσπαση συνεχίζεται αναδρομικά μέχρι τον κόμβο ρίζα. Όταν η διαδικασία διάσπασης φτάσει τελικά τον κόμβο ρίζα, τότε το ύψος του B^+- δένδρου αυξάνεται. Αυτό έχει ως αποτέλεσμα την δημιουργία νέου κόμβου ρίζας, ο οποίος έχει μόνο ένα κλειδί και δύο δείκτες.

Η διαδικασία διαγραφής (deletion process) ενός κλειδιού με τιμή *K*, λειτουργεί με ανάλογο τρόπο. Αμέσως μετά την εύρεση του κατάλληλου κόμβου φύλλου, το κλειδί διαγράφεται από τον κόμβο φύλλο. Παρ'όλα αυτά, εξαιτίας της ιδιότητας ότι κάθε κόμβος πρέπει να έχει τουλάχιστον *d/2* κλειδιά, λαμβάνουν χώρα **επαναζυγιστικές τεχνικές (balancing techniques)** όπως ανακατανομές ή αλληλουχίες (concatenations). Σαν αποτέλεσμα μίας αλληλουχίας, ένας κόμβος μπορεί να αφαιρεθεί από το B^+- δένδρο, και μία αλυσίδα αλληλουχιών (chain of concatenation) μπορεί να μειώσει κατά 1, στην χειρότερη περίπτωση, το ύψος του B^+- δένδρου.

Σε ένα B^+- δένδρο, τάξης *d* με *n* καταχωρίσεις (records), το κόστος ανάκτησης, εισαγωγής ή διαγραφής ενός στοιχείου, είναι ανάλογο του $\log_{d/2} n$, στη χειρότερη περίπτωση. Έτσι, όταν το μέγεθος του κόμβου αυξάνεται, το κόστος των παραπάνω πράξεων πέφτει.

4.3.1 Κατηγορίες B^+- δένδρων για μνήμη φλας

Υπάρχουν δύο βασικές κατηγορίες B^+- δένδρων, κατάλληλα για μνήμη φλας. Πρόκειται για το **B^+- δένδρο (Δίσκου)** (B^+- tree (disk)) και το **B^+- δένδρο (Ημερολογίου)** (B^+- tree (Log)).

Στα B^+- δένδρα (Δίσκου), οι κόμβοι αποθηκεύονται σε πολλαπλές συνεχόμενες σελίδες της μνήμης φλας, ανάλογα με το μέγεθός τους. Για την ανάγνωση ενός κόμβου, πρέπει να αναγνωσθούν οι κατάλληλες σελίδες. Για την ανανέωση ενός κόμβου, διαβάζονται οι κατάλληλες σελίδες στη RAM, τροποποιούνται και στη συνέχεια ξαναγράφονται στη μνήμη φλας.

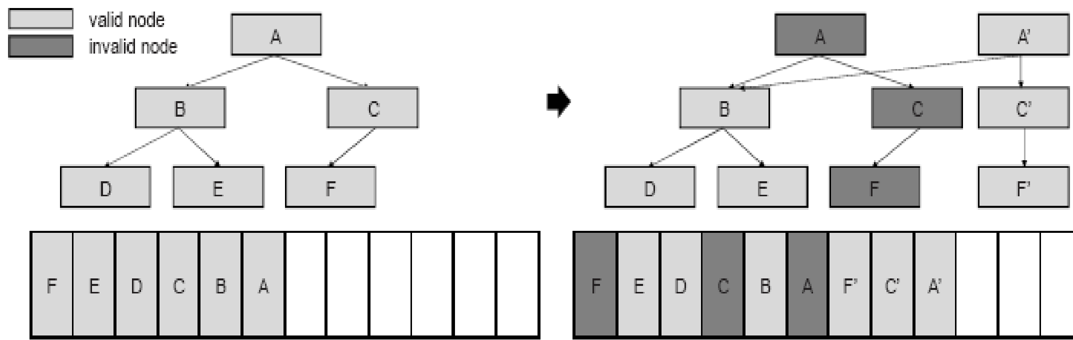
Το βασικό πλεονέκτημα των B^+- δένδρων (Δίσκου), είναι η φορητότητα του κώδικα (code portability). Πιο συγκεκριμένα, η υπάρχουσα υλοποίηση των B^+- δένδρων (Δίσκου) για σκληρούς δίσκους, μπορεί να χρησιμοποιηθεί και στη μνήμη φλας. Το κύριο μειονέκτημά τους είναι το μεγάλο κόστος ανανεώσεων. Ειδικότερα, ακόμα και αν τροποποιηθεί ένα μέρος του κόμβου (π.χ ένας δείκτης), ολόκληρος ο κόμβος πρέπει να διαβαστεί στη RAM και στη συνέχεια να ξαναγραφτεί στη μνήμη φλας. Έτσι, τα B^+- δένδρα (Δίσκου) είναι ακατάλληλα για εφαρμογές που στηρίζονται κατά κύριο λόγο στις εγγραφές (write operations).

Τα B^+- δένδρα (Ημερολογίου), εμπνευσμένα από τα συστήματα αρχείων με δομή ημερολογίου, ελαττώνουν τα υψηλό κόστος ανανεώσεων των B^+- δένδρων (Δίσκου). Η κύρια ιδέα πίσω από τα B^+- δένδρα (Ημερολογίου), είνα η οργάνωση της δομής ευρετηρίου σαν ένα ημερολόγιο. Μία λειτουργία εγγραφής σε έναν κόμβο του δένδρου, κωδικοποιείται σαν μία καταχώριση ημερολογίου, και τοποθετείται σε έναν buffer της μνήμης RAM. Όταν ο buffer περιέχει αρκετά δεδομένα ώστε να γεμίσει μία σελίδα, τότε αυτά γράφονται στη μνήμη φλας.

Το πλεονέκτημα των B^+- δένδρων (Ημερολογίου) είναι το μικρό κόστος ανανέωσης κόμβου, μιάς και το κόστος εγγραφής σελίδας εξοφλείται μέσω πολλαπλών ανανεώσεων. Από την άλλη πλευρά, η ανάγνωση ενός κόμβου είναι δαπανηρή, μιάς και πολλές καταχωρίσεις ημερολογίου, οι οποίες πιθανόν να είναι διασκορπισμένες σε πολλαπλές σελίδες, πρέπει να διαβαστούν για να κατασκευαστεί ο επιθυμητός κόμβος.

4.3.2 Χρήση B^+- δένδρων στο JFFS

Οι σχεδιάστες του JFFSv3, που αποτελεί επέκταση του JFFSv2 που παρουσιάσαμε στο κεφάλαιο 3, υιοθετεί B^+- δένδρα για την οργάνωση καταχωρίσεων καταλόγου (directory entries) και τον εντοπισμό των τελευταίων/πρόσφατων περιεχομένων αρχείων (latest file contents). Παρ'όλα αυτά, εξαιτίας της απουσίας του FTL, μία ανανέωση σε κόμβο φύλλο, οδηγεί σε μία αλυσίδα ανανεώσεων κόμβων ευρετηρίου, μέχρι τον κόμβο ρίζα. Η μέθοδος αυτή ανανέωσης, ονομάζεται **«περιπλανώμενο» δένδρο (wandering tree)**. Ένα δένδρο ονομάζεται «περιπλανώμενο», εάν μία ανανέωση (update) σε ένα κόμβο του δένδρου, απαιτεί την ανανέωση κόμβων πατέρα μέχρι τη ρίζα, εξαιτίας της απουσίας ανανεώσεων εντός θέσης (in-place updates).



Σχήμα 4.11: Ανανέωση (update) σε ένα «περιπλανώμενο» δένδρο.

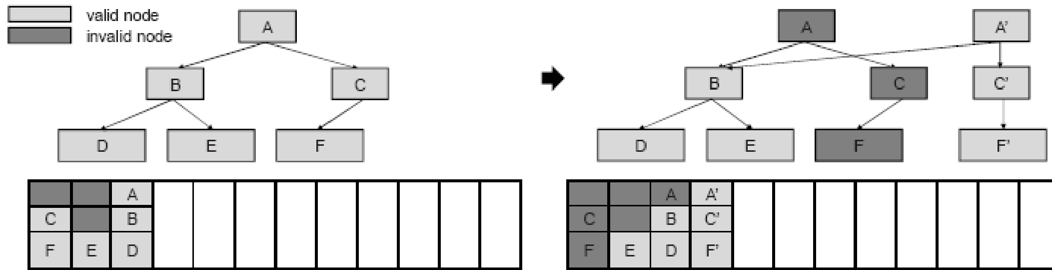
Το σχήμα 4.11 παρουσιάζει ένα παράδειγμα ανανέωσης σε ένα «περιπλανώμενο» δένδρο. Το δένδρο αποτελείται από έξι κόμβους, *A – F*. Εάν πραγματοποιήσουμε μία ανανέωση στον κόμβο *F*, ο τροποποιημένος κόμβος F' πρέπει να γραφτεί σε μία νέα σελίδα, μιας και η μνήμη φλας δεν υποστηρίζει ανανεώσεις εντός θέσης. Ο κόμβος F' παρ'όλα αυτά, δεν είναι προσβάσιμος από την ρίζα διότι ο κόμβος *C* δείχνει ακόμη στον απαρχαιωμένο (obsolete) κόμβο *F*. Έτσι, ο κόμβος *C* ανανεώνεται επίσης, και ο νέος κόμβος C' γράφεται στη μνήμη φλας. Για τον ίδιο ακριβώς λόγο, ο νέος κόμβος ρίζα A', ο οποίος δείχνει στον κόμβο C', γράφεται στη μνήμη φλας.

Για τη μείωση του αριθμού των ανανεώσεων των κόμβων ευρετηρίου, το JFFSv3 χρησιμοποιεί μία δομή δεδομένων που ονομάζεται **δένδρο ημερολογίου (journal tree)** και βρίσκεται στη μνήμη RAM. Όταν πραγματοποιείται μία αλλαγή στο JFFSv3 σύστημα αρχείου, ο αντίστοιχος κόμβος φύλλο γράφεται στη μνήμη φλας, αλλά οι κόμβοι ευρετηρίου ανανεώνονται μόνο στο δένδρο ημερολογίου. Περιοδικά, αυτές οι ανανεώσεις των κόμβων ευρετηρίου, γράφονται στη μνήμη φλας κατά μεγάλες ποσότητες. Το δένδρο ημερολογίου επιτρέπει τη συνένωση πολλών κόμβων ευρετηρίου ενώ παράλληλα ελαττώνει τον αριθμό των εγγραφών στη μνήμη φλας. Παρ'όλα αυτά, η χρήση του δένδρου ημερολογίου δεν είναι κατάλληλη σε μικρά ενσωματωμένα συτήματα περιορισμένης μνήμης. Ακόμα χειρότερα, όσο μεγαλύτερο είναι το μέγεθος του δένδρου ημερολογίου, τόσο περισσότερο αργεί η προσάρτηση (mount) του JFFSv3.

4.4 μ - δένδρα

4.4.1 Ανασκόπηση των μ - δένδρων

Τα μ **- δένδρα**, είναι ισοζυγισμένα (balanced) δένδρα παρόμοια με τα B^+- δένδρα. Είναι σχεδιασμένα τόσο για SLC όσο και για MLC **NAND** μνήμη φλας. Για να αντιμετωπίσουν τα μειονεκτήματα των «περιπλανώμενων» δένδρων, τα μ - δένδρα αποθηκεύουν όλους τους ανανεωμένους κόμβους, κατά μήκος του μονοπατιού από τη ρίζα έως τον κόμβο φύλλο, σε μία μόνο σελίδα της μνήμης φλας. Το σχήμα 4.12 παρουσιάζει τον τρόπο κατά τον οποίο ένα μ - δένδρο μειώνει τον πλήθος των εγγραφών στη μνήμη φλας.

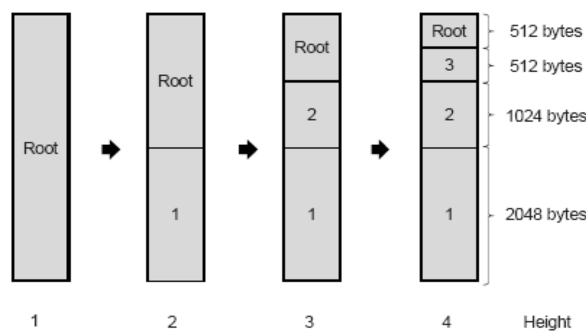


Σχήμα 4.12: Ανανέωση (update) σε ένα μ - δένδρο.

Το πλήθος των ανανεωμένων κόμβων, F', C' και A', είναι ίδιο με αυτό του σχήματος 4.11. Παρ'όλα αυτά, το μ - δένδρο οργανώνει τη διάταξη (layout) της σελίδας κατά τέτοιο τρόπο ώστε, και οι τρεις παραπάνω ανανεωμένοι κόμβοι να αποθηκεύονται σε μία μόνο σελίδα της μνήμης φλας.

Ορίζουμε ως **επίπεδο (level)** ενός κόμβου του μ - δένδρου, το μήκος του μονοπατιού από τα φύλλα προς τον κόμβο + 1. Το επίπεδο ενός κόμβου φύλλου είναι πάντοτε ίσο με τη μονάδα. Επίσης, το **ύψος (height)** ενός μ - δένδρου ορίζεται ως το επίπεδο του κόμβου ρίζα. Για παράδειγμα, το ύψος του δένδρου στο σχήμα 4.12 είναι ίσο με τρία.

Ενώ το μέγεθος των κόμβων σε ένα B^+ - δένδρο είναι καθορισμένο και ίδιο για όλους τους κόμβους, στο μ - δένδρο το μέγεθος κάθε κόμβου ποικίλει, και εξαρτάται από το επίπεδο του κόμβου καθώς και από το ύψος του μ - δένδρου. Αυτό οφείλεται στο γεγονός ότι το μ - δένδρο αποθηκεύει όλους τους ανανεωμένους κόμβους, κατά μήκος του μονοπατιού από τη ρίζα σε οποιοδήποτε φύλλο (A', C' και F' στο σχήμα 4.12), σε μία μόνο σελίδα. Το σχήμα 4.13 που ακολουθεί, παρουσιάζει την μεταβολή της διαμόρφωσης μίας σελίδας (page layout), μεγέθους 4096 bytes.



Σχήμα 4.13: Μεταβολή της διαμόρφωσης σελίδας σύμφωνα με το ύψος του μ - δένδρου.

Ας συμβολίσουμε με H το ύψος του μ - δένδρου και με N_L το σύνολο των κόμβων με επίπεδο L. Σε ένα μ - δένδρο ύψους $H > 2$, ένας κόμβος φύλλο $n \in N_1$ καταλαμβάνει πάντα τη μισή σελίδα. Καθώς το επίπεδο αυξάνεται, το μέγεθος του κόμβου μειώνεται κατά το μισό, όπως φαίνεται και στο σχήμα 4.12. Για έναν κόμβο $m_L \in N_L$, $1 < L < H$, το μέγεθος του κόμβου μειώνεται κατά το ήμισυ,

συγκρινόμενο με το μέγεθος των παιδιών του στο επίπεδο *L*-1. Μόνο ο κόμβος ρίζα έχει το ίδιο μέγεθος με αυτό των παιδιών του. Όταν το μ- δένδρο αποτελείται από ένα μόνο επίπεδο (*H*=1), τότε ολόκληρη η σελίδα ανατίθεται στον κόμβο ρίζα. Η συνολική διαδικασία ευρέσεως ενός κόμβου επιπέδου *L* σε μία σελίδα *P* , σε ένα μ- δένδρο ύψους *H*, παρουσιάζεται στον αλγόριθμο 1 που ακολουθεί.

Algorithm 1 GetNodeFromPage

Input: *page_address P, level L*
Output: *node N*
1: $S \Leftarrow Q/2^L$, where Q is the size of a page
2: $O \Leftarrow S$
3: **if** $L = H$ **then**
4: $\quad S \Leftarrow S * 2$
5: $\quad O \Leftarrow 0$
6: **end if**
7: $N \Leftarrow$ read at page P from offset O with size S
8: **return** N

Η διαμόρφωση σελίδας (page layout) που χρησιμοποιείται στα μ- δένδρα, παρέχει δύο πλεονεκτήματα. Πρώτον, όταν το ύψος του μ- δένδρου αυξάνεται ή μειώνεται, μπορούμε να επαναχρησιμοποιήσουμε όλους τους υπάρχοντες κόμβους, από τη στιγμή που η διαμόρφωση σελίδας εγγυάται ίδιο μέγεθος κόμβου ανά επίπεδο, εκτός από τον κόμβο ρίζα. Εάν διαιρέσουμε την σελίδα ισομερώς μεταξύ των επιπέδων, όταν θα μεταβληθεί το ύψος του δένδρου, θα πραγματοποιηθούν πολλές εγγραφές, μιάς και οι υπάρχοντες κόμβοι δεν «χωράν» στην καινούργια διαμόρφωση σελίδας. Δεύτερον, η διαμόρφωση σελίδας στα μ- δένδρα, αφιερώνει τουλάχιστον τη μισή σελίδα σε κόμβους φύλλα. Το μέγεθος του κόμβου φύλλου, έχει τεράστια επιρροή στην αξιοποίηση αποθηκευτικού χώρου (space utilization). Πιο συγκεκριμένα, όσο μεγαλύτερο είναι το μέγεθος του κόμβου φύλλου, τόσο μειώνεται και ο χωρικός φόρτος (space overhead).

Βασικά, δεν υπάρχουν βασικές διαφορές μεταξύ των B^+- δένδρων και των μ- δένδρων, εκτός από το ότι το μέγεθος ενός κόμβου μ- δένδρου καθορίζεται από το επίπεδό του και από το ύψος του μ- δένδρου.

4.4.2 Διαδικασία ανάκτησης στοιχείου σε μ- δένδρο

Η διαδικασία ανάκτησης (retrieval process) σε ένα μ- δένδρο είναι βασικά ίδια με αυτή των B^+- δένδρων, μιάς και η λογική δομή και των δύο δένδρων είναι πανομοιότυπη. Παρ'όλα αυτά, στα B^+- δένδρα μία σελίδα αποθηκεύει μόνο ένα κόμβο, ενώ στα μ- δένδρα μία σελίδα περιέχει πολλούς κόμβους διαφορετικών επιπέδων. Η συνάρτηση *GetNodeFromPage( )* χρησιμοποιείται επαναληπτικά, κατά τη διάρκεια της διαδικασίας ανάκτησης, για τον εντοπισμό του σωστού κόμβου. Στον αλγόριθμο 2 που ακολουθεί, παρουσιάζεται ο ψευδοκώδικας για την ανάκτηση μίας καταχώρισης (record) που αντιστοιχεί στο κλειδί *K*.

Algorithm 2 Retrieval

Input: *key* K (search predicate)
Output: *page_address* O (which points to the record corresponding to K)
1: $C \Leftarrow$ GetNodeFromPage(root page address, H)
2: $L \Leftarrow H$
3: **while** $C.type \neq LEAF$ **do**
4: $K_i \Leftarrow$ smallest search-key greater than K
5: $L \Leftarrow L - 1$
6: **if** K_i exists **then**
7: $C \Leftarrow$ GetNodeFromPage(P_i, L)
8: **else**
9: $C \Leftarrow$ GetNodeFromPage(P_m, L), where m is the number of pointers in C
10: **end if**
11: **end while**
12: **if** K_i exists in C, such that $K_i = K$ **then**
13: **return** P_i
14: **else**
15: **return** $NULL$
16: **end if**

4.4.3 Διαδικασία εισαγωγής στοιχείου σε μ - δένδρο

Ο αλγόριθμος 3 που παρουσιάζεται παρακάτω, περιλαμβάνει τον ψευδοκώδικα για την εισαγωγή ενός κλειδιού *K* και της διεύθυνσης *P* που αντιστοιχεί στην καταχώριση.

Algorithm 3 Insertion

Input: *key* K, *page_address* P (which points to the record corresponding to K)
1: allocate a new page N
2: $(R, K', P') \Leftarrow$ InsertEntry(K, P, N, root page address, H)
3: **if** $R = FULL$ **then**
4: allocate a new page N'
5: $C \Leftarrow$ GetNodeFromPage(N, H)
6: $H \Leftarrow H + 1$
7: $(C_l, C_r) =$ Split(C)
8: $C' \Leftarrow$ GetNodeFromPage(N, H)
9: insert $(C_l.K_1, N)$ and $(C_r.K_1, N')$ into C'
10: write node C_l on page N
11: write node C_r on page N'
12: write node C' on page N'
13: **end if**

Η συνάρτηση *Insertion( )* δεσμεύει αρχικά μία νέα σελίδα *N* για να αντιγράψει όλους τους κόμβους που επρόκειτο να ανανεωθούν. Στη συνέχεια καλεί την συνάρτηση *InsertEntry( )*(αλγόριθμος 4) για την εισαγωγή της καταχώρισης (*K, P*) στην σελίδα *N*.

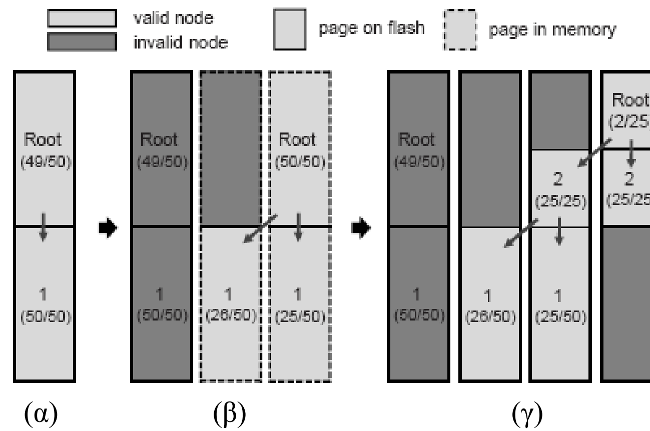
Algorithm 4 InsertEntry

```
Input: key K, page_address P, N, B, level L
Output: return_value R, key K', page_address P'
 1: C ⇐ GetNodeFromPage(B, L)
 2: if C.type ≠ LEAF then
 3:    find C.P_i, such that C.K_i ≤ K < C.K_{i+1}
 4:    if C.P_i doesn't exist then
 5:       i ⇐ m, where m is the number of pointers in C
 6:    end if
 7:    (R, K', P') ⇐ InsertEntry(K, P, N, C.P_i, L − 1)
 8:    C.P_i ⇐ N
 9:    if R = SPLIT then
10:       K ⇐ K', P ⇐ P', N ⇐ P'
11:    else
12:       write node C on page N
13:       return R ⇐ NULL
14:    end if
15: end if
16: if C has space for (K, P) then
17:    insert (K, P) into C
18:    write node C on page N
19:    if C is full then
20:       return R ⇐ FULL
21:    else
22:       return R ⇐ NULL
23:    end if
24: else
25:    allocate a new page N'
26:    (C_l, C_r) ⇐ Split(C)
27:    insert (K, P) into (C_r.K_1 > K)? C_r : C_l
28:    if C_l.type ≠ LEAF & ∃C_r.P_i = N then
29:       swap C_l ⇔ C_r
30:    end if
31:    write node C_l on page N
32:    write node C_r on page N'
33:    return R ⇐ SPLIT, K' ⇐ C_r.K_1, P' ⇐ N'
34: end if
```

Με το τέλος της κλήσης, η *Insertion( )* ελέγχει την τιμή επιστροφής της συνάρτησης *InsertEntry( )*. Εάν η τιμή επιστοφής είναι ίση με *FULL*, τότε ο κόμβος ρίζα έχει γεμίσει, ως αποτέλεσμα της τρέχουσας εισαγωγής, και το ύψος του μ-δένδρου αυξάνεται κατά ένα.

Όπως και στα B^{+}- δένδρα, η συνάρτηση *InsertEntry( )* επισκέπτεται επαναληπτικά πολλούς κόμβους, από τη ρίζα έως τα φύλλα του δένδρου. Κανονικά, εισάγει την καταχώριση (*K, P*) σε έναν κόμβο φύλλο και αντιγράφει όλους τους κόμβους από το φύλλο προς τη ρίζα, στη σελίδα *N*. Εάν ο κόμβος φύλλο είναι ήδη γεμάτος, τότε δεσμεύεται μία νέα σελίδα $N^{'}$και πραγματοποιείται και διάσπαση (split) του συγκεκριμένου κόμβου φύλλου. Πιο συγκεκριμένα, ο αρχικός κόμβος φύλλο *C*, διασπάται στους κόμβους C_lκαι C_r, οι οποίοι αποθηκεύονται στις σελίδες *N* και $N^{'}$αντίστοιχα. Η νέα καταχώριση εισάγεται είτε στον κόμβο C_l είτε στον κόμβο C_r, ανάλογα με την τιμή *K* του κλειδιού. Τέλος, η καταχώριση ($C_r.K_1, N^{'}$) εισάγεται στον αντίστοιχο κόμβο πατέρα. Εάν ο κόμβος πατέρα, γεμίσει εξαιτίας της παραπάνω εισαγωγής, η διαδικασία διάσπασης λαμβάνει χώρα ξανά.

Όταν ο κόμβος ρίζα γεμίσει, η διαδικασία εισαγωγής αυξάνει το ύψος του μ- δένδρου, όπως φαίνεται στις γραμμές 4 - 12 του αλγορίθμου 3. Το σχήμα 4.14 που ακολουθεί, παρουσιάζει ένα παράδειγμα εισαγωγής στο οποίο πραγματοποιείται αύξηση του ύψους του μ- δένδρου. Στο σχήμα, ο αριθμός σε κάθε κόμβο προσδιορίζει το επίπεδο (level) του κόμβου, ενώ με (*a/b*) υποδηλώνεται ότι ο κόμβος «στεγάζει» *a* καταχωρίσεις από τις *b* που μπορεί να αποθηκεύσει συνολικά. Το βέλος μεταξύ δύο κόμβων, υποδηλώνει ότι ο ένας κόμβος διαθέτει έναν δείκτη προς τον άλλον κόμβο.



Σχήμα 4.14: Παράδειγμα εισαγωγής ενός στοιχείου σε μ - δένδρο.

Η αρχική κατάσταση του μ- δένδρου πριν την εισαγωγή του νέου στοιχείου, φαίνεται στο σχήμα 4.14 (α), όπου ο κόμβος φύλλο είναι γεμάτος και ο κόμβος ρίζα έχει χώρο για μία ακόμη καταχώριση. Υποθέτουμε ότι το νέο στοιχείο εισάγεται στον κόμβο φύλλο. Η κλήση της *InsertEntry( ),* έχει ως αποτέλεσμα την διάσπαση του κόμβου φύλλου και την εισαγωγή ενός στοιχείου στον κόμβο ρίζα. Το σχήμα 4.14 (β) παρουσιάζει την ενδιάμεση κατάσταση μετά την επιστροφή της *InsertEntry( ).* Επειδή ο κόμβος ρίζα γεμίζει, πραγματοποιείται διάσπαση της ρίζας σε δύο νέους κόμβους χαμηλώτερου επιπέδου, με αποτέλεσμα το ύψος του μ- δένδρου να αυξάνεται κατά ένα.

Αξίζει να αναφέρουμε ότι στις γραμμές 28 – 30 του αλγορίθμου 4, δύο κόμβοι C_l και C_r, που προκύπτουν απο τη διάσπαση του κόμβου *C,* ανταλλάσσουν τα περιεχόμενά τους, όταν κάποιο στοιχείο/καταχώριση του κόμβου C_r έχει ένα δείκτη προς ένα κόμβο της σελίδας *N.* Αυτό συμβαίνει διότι στα μ- δένδρα, μόνο ο απευθείας κόμβος παιδί μπορεί να αποθηκευτεί στο αμέσως χαμηλότερο επίπεδο, μέσα στην ίδια σελίδα. Η ιδιότητα αυτή ονομάζεται σχέση απογόνου – προγόνου (descendant – ancestor relationship), η οποία διευκολύνει την διαδικασία συλλογής απορριμμάτων.

4.4.4 Διαδικασία διαγραφής στοιχείου σε μ - δένδρο

Ο αλγόριθμος 5 παρουσιάζει την διαδικασία διαγραφής στοιχείου σε ένα μ - δένδρο. Η συνάρτηση *Deletion( )* είναι παρόμοια με την συνάρτηση *Insertion( ),* με τη μόνη

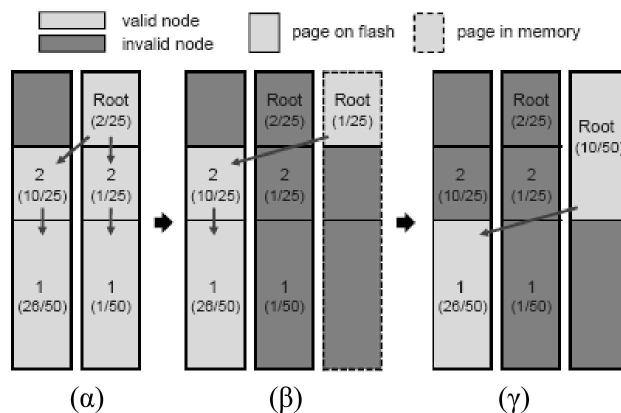
διαφορά ότι το ύψος του δένδρου μπορεί να μειωθεί κατά ένα σαν αποτέλεσμα της διαγραφής κάποιου στοιχείου.

Algorithm 5 Deletion

Input: *key* K
1: allocate a new page N
2: $R \Leftarrow$ DeleteEntry(K, N, root page address, H)
3: **if** $R = ONE$ **then**
4: $\quad C \Leftarrow$ GetNodeFromPage(N, H)
5: $\quad C' \Leftarrow$ GetNodeFromPage($C.P_1$, H)
6: $\quad H \Leftarrow H - 1$
7: $\quad C \Leftarrow$ GetNodeFromPage(N, H)
8: $\quad C \Leftarrow C'$
9: $\quad$ write node C' on page N
10: **end if**

Η συνάρτηση *DeleteEntry( )* επισκέπτεται διαδοχικούς κόμβους για να εντοπίσει την προς – διαγραφή καταχώριση, που αντιστοιχεί συο κλειδί *K*. Εάν ο κόμβος φύλλο αδειάσει εξαιτίας της διαγραφής, η καταχώριση στον κόμβο πατέρα, η οποία δείχνει προς τον άδειο κόμβο φύλλο, αφαιρείται επίσης. Η διαδικασία αυτή της διαγραφής, μπορεί να προωθηθεί έως και στον κόμβο ρίζα του δένδρου.

Όπως φαίνεται και στις γραμμές 4 – 9 του αλγορίθμου 5, το μ - δένδρο μειώνει το ύψος του όταν ο κόμβος ρίζα έχει μόνο δείκτη. Το σχήμα 4.15 που ακολουθεί, παρουσιάζει ένα παράδειγμα διαγραφής στοιχείου, στο οποίο πραγματοποιείται και μείωση του ύψους του μ - δένδρου.



Σχήμα 4.15: Παράδειγμα διαγραφής ενός στοιχείου σε μ - δένδρο.

Το σχήμα 4.15 (α) δείχνει την αρχική κατάσταση του μ - δένδρου πριν τη διαγραφή ενός στοιχείου. Υποθέτουμε ότι διαγράφουμε ένα στοιχείο/καταχώριση που είναι αποθηκευμένο στον δεξιό κόμβο φύλλο του δένδρου, όπως φαίνεται στο σχήμα 4.15 (α). Αυτό έχει ως αποτέλεσμα, ο κόμβος ρίζα να διαθέτει μόνο έναν δείκτη, όπως φαίνεται στο σχήμα 4.15 (β). Η τελική κατάσταση παρουσιάζεται στο σχήμα 4.15 (γ), όπου ο κόμβος επιπέδου 2 προβιβάζεται στον κόμβο ρίζα, και το ύψος του μ - δένδρου μειώνεται κατά ένα.

4.4.5 Ανάλυση πολυπλοκότητας μ - δένδρων

Στην ενότητα αυτή, συγκρίνουμε τη συμπεριφορά των B^+- δένδρων και των μ- δένδρων. Προς ευκολία της ανάλυσης, υποθέτουμε ότι το σύστημα δεν διαθέτει κρυφή μνήμη (cache memory) και ότι υπάρχουν αρκετά ελεύθερα μπλοκ για την πραγματοποίηση τριών πράξεων/λειτουργιών, χωρίς την εμφάνιση της διαδικασίας συλλογής απορριμμάτων. Ο πίνακας 4.3 που ακολουθεί, παρουσιάζει τους συμβολισμούς που χρησιμοποιούνται κατά την ανάλυση.

Σύμβολο	**Ορισμός**
h_B, h_μ	Ύψος B^+ - δένδρου (h_B) και μ - δένδρου (h_μ).
l	Επίπεδο ενός κόμβου.
d_l	Μέγιστος αριθμός στοιχείων ενός κόμβου επιπέδου *l*.
f	Μέγιστος αριθμός στοιχείων μίας σελίδας.
n_h	Σύνολο καταχωρίσεων (records) δένδρου ύψους *h*.
u_B, u_μ	Συνολικός αριθμός έγκυρων σελίδων σε B^+ - δένδρο (u_B) και σε μ - δένδρο (u_μ)
c_r	Κόστος ανάγνωσης στη μνήμη φλας.
c_w	Κόστος εγγραφής στη μνήμη φλας.

Πίνακας 4.3: Συμβολισμοί ανάλυσης.

4.4.5.1 Κόστος πράξεων

Ο πίνακας 4.4 παρουσιάζει το κόστος των τριών πράξεων που πραγματοποιούνται στα δένδρα, όπου δεν υπάρχει διάσπαση κόμβου ή διαγραφή κόμβου.

Operations	B^+-Tree	μ-Tree
Retrieval	$c_r h_B$	$c_r h_\mu$
Insertion	$(c_r + c_w) h_B$	$c_r h_\mu + c_w$
Deletion	$(c_r + c_w) h_B$	$c_r h_\mu + c_w$

Πίνακας 4.4: Κόστος πράξεων.

Όπως φαίνεται και στον παραπάνω πίνακα, το κόστος ανάκτησης ενός στοιχείου είναι ανάλογο του ύψους των δένδρων. Για τις πράξεις εισαγωγής και διαγραφής, πρέπει πρώτα να φτάσουμε σε ένα κόμβο φύλλο, και στη συνέχεια ανανεώνεται το συγκεκριμένο φύλλο. Το B^+- δένδρο απαιτεί τόσες λειτουργίες εγγραφής όσες και το ύψος του, ενώ το μ - δένδρο απαιτεί μόνο μία λειτουργία εγγραφής.

4.4.5.2 Το ύψος των δένδρων

Το ύψος του μ - δένδρου μπορεί να είναι ελαφρώς μεγαλύτερο από αυτό του B^+- δένδρου, για τον ίδιο αριθμό στοιχείων. Αυτό οφείλεται στο γεγονός ότι, καθώς το

επίπεδο (level) αυξάνεται, ο μέγιστος βαθμός εξόδου (fanout) ενός κόμβου μειώνεται σε ένα μ - δένδρο, μιάς και το μέγεθος του κόμβου μικραίνει.

Πρώτα θα αναλύσουμε το ύψος ενός B^+ - δένδρου, στο οποίο το μέγεθος των κόμβων είναι ίδιο για όλους τους κόμβους. Καθώς κάθε κόμβος αποθηκεύεται σε μία ξεχωριστή σελίδα, έχουμε:

$$d_l = f \text{ , για } 1 \le l \le h_B \quad (1)$$

Ο αριθμός n_h υπολογίζεται από τον παρακάτω τύπο:

$$n_h = \prod_{i=1}^{h} d_i = f^h \qquad (2)$$

Από τη σχέση (2), το ύψος (h_B) του B^+ - δένδρου, εκφράζεται όπως παρακάτω, για *n* καταχωρίσεις:

$$h_B = \log_f n \qquad (3)$$

Σε ένα μ - δένδρο, ο αριθμός d_l δίνεται από τον τύπο:

$$d_l = \begin{cases} f / 2^{l-1} (l = h_\mu) \\ f / 2^l (l < h_\mu) \end{cases} \qquad (4)$$

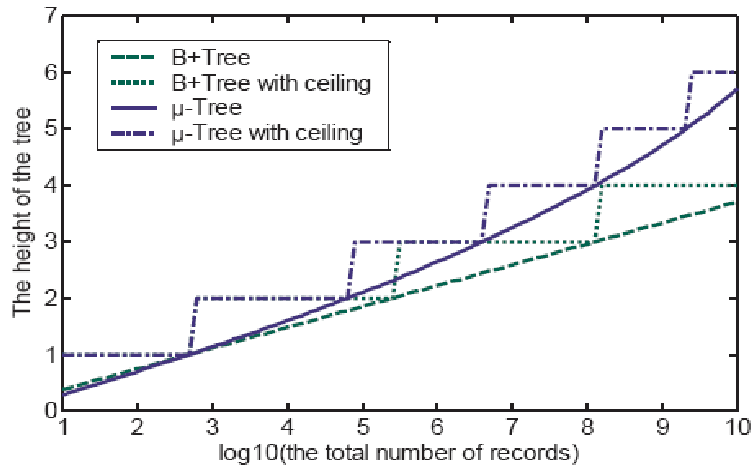
Ενώ ο αριθμός n_h υπολογίζεται από τον παρακάτω τύπο:

$$n_h = \prod_{i=1}^{h} d_i = 2 \prod_{i=1}^{h} (f / 2^i) \quad (5)$$

Από τη σχέση (5), το ύψος h_μ ενός μ - δένδρου με *n* καταχωρίσεις, αναπαρίσταται όπως παρακάτω:

$$h_\mu = -\log_2 \frac{\sqrt{2}}{f} - \sqrt{\log_2^2 \frac{\sqrt{2}}{f} - 2\log_2 \frac{n}{2}} \qquad (6)$$

Το σχήμα 4.16 που ακολουθεί, παρουσιάζει το ύψος h_B, $\lceil h_B \rceil$, h_μ και $\left| h_\mu \right|$ ενός B^+ - δένδρου και ενός μ - δένδρου, όταν το μέγεθος σελίδας είναι 4Kbytes και το μέγεθος της καταχώρισης 8 bytes (δηλ. f=512).



Σχήμα 4.16: Σύγκριση ύψους B^+ - δένδρου και μ - δένδρου.

Παρατηρούμε ότι το ύψος του μ - δένδρου είναι ίσο ή το πολύ κατά ένα επίπεδο μεγαλύτερο από αυτό του B^+ - δένδρου, για ένα δισεκατομμύριο καταχωρίσεις. Παρ'όλο που το μεγαλύτερο ύψος ενός δένδρου αυξάνει τον αριθμό των αναγνώσεων στη μνήμη φλας, κατά τη διάρκεια ανάκτησης ενός στοιχείου, η επίπτωση στη συνολική απόδοση του συστήματος, μπορεί να μετριαστεί από το γεγονός ότι οι λειτουργίες ανάγνωσης είναι πολύ γρηγορότερες από τις λειτουργίες εγγραφής, στην NAND μνήμη φλας.

4.4.5.3 Χωρικός φόρτος (Space overhead)

Τα μ - δένδρα απαιτούν περισσότερο αποθηκευτικό χώρο σε σύγκριση με τα B^+ - δένδρα, για τον ίδιο αριθμό στοιχείων/καταχωρίσεων, μιάς και ο μέγιστος βαθμός εξόδου (fanout) των κόμβων φύλλων ενός μ - δένδρου είναι μικρότερος από αυτόν του B^+ - δένδρου. Αν υποθέσουμε ότι κάθε κόμβος φύλλο είναι γεμάτος, ο αριθμός u_B ενός B^+ - δένδρου, δίνεται από τον παρακάτω τύπο:

$$u_B = \left\lceil \frac{n}{d_1} \right\rceil + \left\lceil \frac{n}{d_1 d_2} \right\rceil + \ldots + \left\lceil \frac{n}{d_1 d_2 \ldots d_{h_B}} \right\rceil \quad (7)$$

Κάνοντας χρήση της σχέσης (1), έχουμε:

$$u_B = \left\lceil \frac{n}{f} \right\rceil + \left\lceil \frac{n}{f^2} \right\rceil + \ldots + \left\lceil \frac{n}{f^{h_B}} \right\rceil \quad (8)$$

Στις περισσότερες περιπτώσεις ο αριθμός f είναι αρκετά μεγάλος (f >100), με αποτέλεσμα να αγνοούμε τον όρο $\left\lceil \frac{n}{f^2} \right\rceil + \ldots + \left\lceil \frac{n}{f^{h_B}} \right\rceil$ της σχέσης (8). Έτσι, ο αριθμός u_B ενός B^+ - δένδρου, δίνεται από τον τύπο:

$$u_B \approx \left\lceil \frac{n}{f} \right\rceil \qquad (9)$$

Στα μ - δένδρα, μετράμε μόνο τους κόμβους φύλλα μιάς και οι κόμβοι ευρετηρίου (index nodes) αποθηκεύονται μαζί με τους κόμβους φύλλα. Έτσι, ο αριθμός u_μ ενός - μ -δένδρου, δίνεται από τον τύπο:

$$u_\mu = \left\lceil \frac{n}{d_1} \right\rceil = \left\lceil \frac{n}{f/2} \right\rceil \qquad (10)$$

Από τις σχέσεις (9) και (10), προκύπτει:

$$u_B \approx \frac{u_\mu}{2} \qquad (11)$$

Τα μ - δένδρα καταλαμβάνουν κατά προσέγγιση, διπλάσιο αριθμό σελίδων μνήμης φλας σε σχέση με τα B^+ - δένδρα. Ο χωρικός φόρτος των μ - δένδρων μπορεί να ελαττωθεί, αναθέτοντας περισσότερο αποθηκευτικό χώρο σε κόμβους φύλλα, στα πλαίσια της σελίδας. Αξίζει να σημειώσουμε ότι τα B^+- δένδρα, παράγουν περισσότερες μη έγκυρες σελίδες, σε σχέση με τα μ - δένδρα, κατά τη διάρκεια εισαγωγών και διαγραφών στοιχείων.

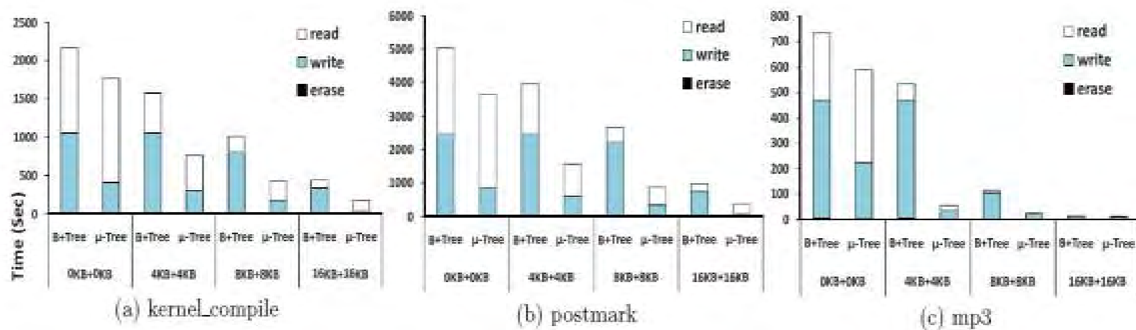
4.4.6 Αποτελέσματα πειραμάτων

Πριν την παρουσίαση των αποτελεσμάτων των πειραμάτων σχετικά με την απόδοση των μ - δένδρων, πρέπει να αναφέρουμε τη χρήση κρυφής μνήμης στα μ - δένδρα. Πιο συγκεκριμένα, χρησιμοποιείται ένα σύστημα κρυφής μνήμης το οποίο απαρτίζεται από μία κρυφή μνήμη ανάγνωσης (read cache) και μία κρυφή μνήμη εγγραφής (write cache), οι οποίες αποθηκεύουν αμοιβαίως αποκλειστικές σελίδες. Όταν πραγματοποιούνται λειτουργίες ανάγνωσης σε ένα μ - δένδρο, το σύστημα κρυφής μνήμης εξετάζει πρώτα την κρυφή μνήμη ανάγνωσης και στη συνέχεια την κρυφή μνήμη εγγαφής. Εάν η ζητούμενη σελίδα δεν υπάρχει σε καμία από τις δύο κρυφές μνήμες, τότε στέλνεται στη μνήμη φλας μία εντολή για ανάγνωση. Όταν το μ - δένδρο γράφει μία νέα σελίδα, τότε πραγματοποιείται μία αίτηση ανάθεσης σελίδας στην κρυφή μνήμη εγγραφής. Εάν η κρυφή μνήμη εγγραφής είναι γεμάτη, τότε όλες οι σελίδες που περιέχει, γράφονται στη μνήμη φλας.

Για την εκτίμηση της απόδοσης των μ - δένδρων, πραγματοποιήθηκαν πειράματα με πραγματικό φόρτο εργασίας (workload), τα χαρακτηριστικά του οποίου φαίνονται στον πίνακα που ακολουθεί. Για την εξαγωγή συμπερασμάτων, τα πειράματα πραγματοποιήθηκαν και σε B^+- δένδρα και τα αποτελέσματα φαίνονται στα παρακάτω σχήματα. Αξίζει στο σημείο αυτό να αναφέρουμε ότι, τα πειράματα πραγματοποιήθηκαν τόσο χωρίς όσο και με την παρουσία κρυφής μνήμης ανάγνωσης και εγγραφής (4KB+4KB, 8KB+8KB, 16KB+16KB).

Trace	Description	Retrieval	Insertion	Deletion
kernel_compile	This trace is obtained while we untar, compile, clean, and remove the Linux kernel source.	2,274,867	260,974	236,027
postmark	This is a trace for postmark benchmark. This benchmark models the workload of an e-mail server, where lots of files are created and deleted.	4,617,494	574,148	391,847
mp3	This is a trace for synthetic workload, where we model the usage scenario of a MP3 player. The storage of 8 Gbytes is filled with MP3 files fully, and then 50% of files are deleted and copied again five times.	512,986	223,188	23,950

Πίνακας 4.5: Χαρακτηριστικά του φόρτου εργασίας των πειραμάτων.



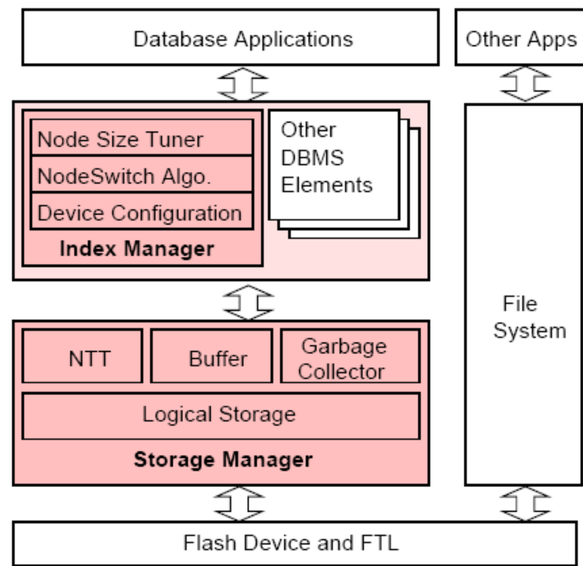
Σχήμα 4.17: Αποτελέσματα πειραμάτων.

Παρ'όλο που το μ- δένδρο πραγματοποιεί λίγες παραπάνω λειτουργίες ανάγνωσης μνήμης φλας σε σχέση με το B^+- δένδρο, το βασικό του πλεονέκτημα είναι ο αισθητά μικρότερος αριθμός λειτουργιών εγγραφής, γεγονός που οδηγεί το μ - δένδρο στην ταχύτερη εκτέλεση των αιτούμενων πράξεων.

4.5 FlashDB

Το FlashDB είναι μία βάση δεδομένων κατάλληλη για συσκευές που χρησιμοποιούν μνήμη φλας. Είναι **αυτορυθμιζόμενη (self-tuning)**, με την έννοια ότι αφού αρχικά συντονιστεί/διαμορφωθεί με τα κόστη ανάγνωσης και εγγραφής σελίδας της υπάρχουσας αποθηκευτικής συσκευής, στη συνέχεια προσαρμόζει αυτόματα την αποθηκευτική δομή της (storage structure), με τέτοιο τρόπο ώστε να ελαχιστοποιεί την κατανάλωση ενέργειας και την καθυστέρηση, που σχετίζονται με το φόρτο εργασίας (workload) που της έχει ανατεθεί. Ανάλογα με το τύπο της φλας συσκευής, η βάση δεδομένων FlashDB οργανώνει, με διαφορετικό τρόπο κάθε φορά, τα δεδομένα στην υπάρχουσα αποθηκευτική συσκευή. Η FlashDB αποτελείται από δύο βασικά συστατικά (σχήμα 4.18): το **σύστημα διαχείρισης βάσεως δεδομένων (Database Management System)**, το οποίο υλοποιεί τις λειτουργίες της βάσης δεδομένων, όπως για παράδειγμα διαχείριση ευρετηρίου (index management) και συλλογή επερώτησεων, καθώς και τον **διαχειριστή αποθήκευσης (Storage Manager)** ο οποίος ασχολείται με την προσωρινή αποθήκευση δεδομένων (data buffering) και τη συλλογή απορριμμάτων (garbage collection). Στις ενότητες που ακολουθούν, θα παρουσιαστεί ο αυτορυθμιζόμενος διαχειριστής ευρετηρίου (self-tuning Index Manager), ο οποίος χρησιμοποιεί μία δομή B^+- δένδρου, και αποτελεί

μέρος του συστήματος διαχείρισης βάσεως δεδομένων. Επίσης θα παρουσιαστούν οι λειτουργίες του διαχειριστή αποθήκευσης (Storage Manager).



Σχήμα 4.18: Αρχιτεκτονική της βάσης δεδομένων FlashDB.

4.5.1 Αυτορυθμιζόμενη ευρετηρίαση (Self-tuning indexing)

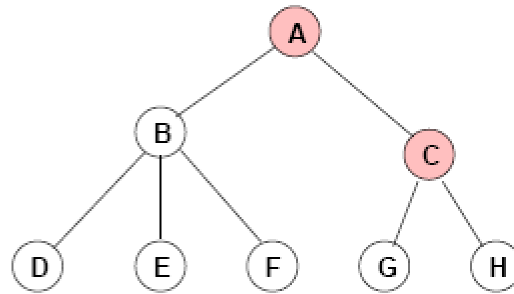
Στην ενότητα αυτή, θα παρουσιάσουμε τα αυτορυθμιζόμενα B^+- δένδρα, τα οποία ονομάζονται B^+- δένδρα(ST), και είναι κατάλληλα για NAND μνήμη φλας.

4.5.1.1 Σχεδίαση B^+- δένδρων(ST)

Το καινούργιο χαρακτηριστικό των B^+- δένδρων(ST) είναι η δυνατότητα να αποθηκεύουν ένα κόμβο είτε σε κατάσταση δίσκου (Disk mode) είτε σε κατάσταση ημερολογίου (Log mode). Όταν ένας κόμβος βρίσκεται σε κατάσταση ημερολογίου, κάθε λειτουργία ανανέωσης του συγκεκριμένου κόμβου (π.χ προσθήκη ή διαγραφή κλειδιού), γράφεται σαν μία ξεχωριστή καταχώριση ημερολογίου (log entry), όπως και στα B^+- δένδρα(Ημερολογίου). Έτσι, για να διαβάσουμε ένα κόμβο που βρίσκεται σε κατάσταση ημερολογίου, πρέπει να διαβαστούν όλες οι καταχωρίσεις ημερολογίου του κόμβου (που πιθανόν να είναι διασκορπισμένες σε πολλές σελίδες). Όταν ένας κόμβος βρίσκεται σε κατάσταση δίσκου, ολόκληρος ο κόμβος γράφεται μαζί σε συνεχόμενες σελίδες (ο αριθμός των σελίδων εξαρτάται από το μέγεθος των κόμβων). Κατ'αυτόν τον τρόπο, η ανάγνωση ενός κόμβου απαιτεί την ανάγνωση των αντίστοιχων τομέων.

Οποιαδήποτε χρονική στιγμή, μερικοί λογικοί κόμβοι του B^+- δένδρου(ST) μπορεί να βρίσκονται σε κατάσταση ημερολογίου, ενώ κάποιοι άλλοι σε κατάσταση δίσκου. Ειδικότερα, οι κόμβοι μπορεί να αλλάξουν την κατάστασή τους δυναμικά, καθώς μεταβάλλεται ο φόρτος εγρασίας (workload) ή η αποθηκευτική συσκευή. Το σχήμα 4.19 παρουσιάζει ένα στιγμυότυπο ενός B^+- δένδρου(ST), όπου με ροζ χρώμα

αναπαρίστανται οι κόμβοι που είναι σε κατάσταση δίσκου, ενώ οι υπόλοιποι βρίσκονται σε κατάσταση ημερολογίου.

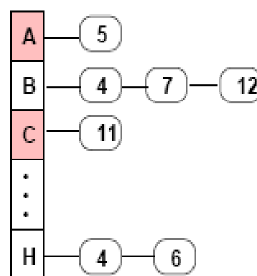


Σχήμα 4.19: Ένα λογικό B^+ - δένδρο(ST).

4.5.1.1.1 Συστατικά διαχειριστή αποθήκευσης

Τα B^+ - δένδρα(ST) εισάγουν δύο συστατικά στον διαχειριστή αποθήκευσης (Storage Manager) της FlashDB: μία **περιοχή προσωρινής αποθήκευσης ημερολογίου (Log Buffer)** και έναν **πίνακα μετάφρασης κόμβων (Node Translation Table - NTT)**. Η περιοχή προσωρινής αποθήκευσης ημερολογίου, η οποία αποθηκεύει δεδομένα σε ποσότητα ίση με το μέγεθος ενός τομέα (sector), χρησιμοποιείται από κόμβους που βρίσκονται σε κατάσταση ημερολογίου (Log mode). Όταν ένας κόμβος σε κατάσταση ημερολογίου τροποποιείται, οι αντιστοίχες καταχωρίσεις ημερολογίου αποθηκεύονται προσωρινά στην περιοχή προσωρινής αποθήκευσης ημερολογίου. Όταν η περιοχή προσωρινής αποθήκευσης ημερολογίου περιέχει καταχωρίσιες σε ποσότητα ίση με το μέγεθος μίας σελίδας, τότε αυτές οι καταχωρίσεις γράφονται μαζί στη μνήμη φλας, γλιτώνοντας έτσι τις δαπανηρές μικρές εγγραφές.

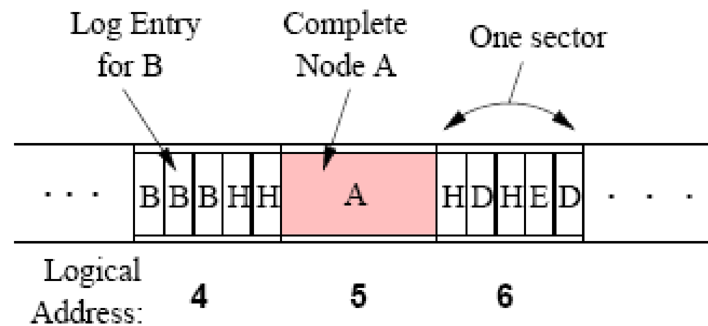
Ο πίνακας μετάφρασης κόμβων αντιστοιχίζει λογικούς κόμβους B^+ - δένδρου(ST) στην τρέχουσα κατάστασή τους (δίσκου ή ημερολογίου) και στην φυσική τους αναπαράσταση. Το σχήμα 4.20 παρουσιάζει ένα μέρος του πίνακα μετάφρασης κόμβων που αντιστοιχεί στο δένδρο του σχήματος 4.19.



Σχήμα 4.20: Πίνακας μετάφρασης κόμβων.

Για ένα κόμβο που βρίσκεται σε κατάσταση δίσκου, για παράδειγμα ο κόμβος A στο παραπάνω σχήμα, ο πίνακας μετάφρασης κόμβων αποθηκεύει την διεύθυνση του τομέα της μνήμης φλας, στον οποίο είναι αποθηκευμένος ο κόμβος. Εδώ ο κόμβος A

είναι αποθηκευμένος στον τομέα 5 της μνήμης φλας. Για ένα κόμβο που βρίσκεται σε κατάσταση ημερολογίου, ο πίνακας μετάφρασης κόμβων διατηρεί μία συνδεδεμένη λίστα από διευθύνσεις όλων των τομέων, που περιέχουν τουλάχιστον μία έγκυρη καταχώριση ημερολογίου για αυτόν τον κόμβο. Για παράδειγμα, στο παραπάνω σχήμα, ο κόμβος B έχει τουλάχιστον μία καταχώριση ημερολογίου σε κάθε έναν από τους τομείς 4, 7 και 12. Στο σημείο αυτό πρέπει να σημειώσουμε ότι ένας τομέας που περιέχει καταχωρίσεις για τον κόμβο B για παράδειγμα, μπορεί να περιέχει καταχωρίσεις και για άλλους κόμβους (π.χ ο τομέας 4 στο σχήμα 4.21), μιας και η περιοχή προσωρινής αποθήκευσης ημερολογίου περιέχει καταχωρίσεις διαφορετικών κόμβων όταν τα περιεχόμενα αυτά μεταφέρονται στη μνήμη φλας.



Σχήμα 4.21: Τομείς της μνήμης φλας. Ο τομέας 5 αποθηκεύει μόνο τον κόμβο A που βρίσκεται σε κατάσταση δίσκου. Οι τομείς 4 και 6 περιέχουν καταχωρίσεις ημερολογίου για διαφορετικούς κόμβους που βρίσκονται σε κατάσταση ημερολογίου.

4.5.1.1.2 Βασικές πράξεις

Για τη δημιουργία ενός κόμβου με αναγνωριστικό (id) *x*, κατασκευάζεται μία καταχώριση με αναγνωριστικό *x* και με κατάσταση ημερολογίου, στον πίνακα μετάφρασης κόμβων. Για την ανάγνωση ή την ανανέωση ενός κόμβου *x,* διαβάζεται σε πρώτη φάση η κατάσταση του κόμβου από τον πίνακα μετάφρασης κόμβων (*NTT[x]*). Εάν ο κόμβος *x* βρίσκεται σε κατάσταση δίσκου, τότε ο κόμβος διαβάζεται (ανανεώνεται) από τον (στον) τομέα που υποδεικνύει ο πίνακα μετάφρασης κόμβων. Εάν ο κόμβος *x* βρίσκεται σε κατάσταση ημερολογίου τα πράγματα είναι λίγο πιο περίπλοκα. Πιο συγκεκριμένα, για την ανανέωση του κόμβου *x,* κατασκευάζεται μία καταχώριση ημερολογίου που αντικατοπτρίζει την πράξη της ανανέωσης, και στη συνέχεια αποθηκεύεται στην περιοχή προσωρινής αποθήκευσης ημερολογίου (Log Buffer). Αργότερα, όταν η περιοχή προσωρινής αποθήκευσης ημερολογίου περιέχει καταχωρίσιες σε ποσότητα ίση με το μέγεθος ενός τομέα, τότε αυτές οι καταχωρίσεις γράφονται σε ένα διαθέσιμο τομέα της μνήμης φλας, και η διεύθυνση του τομέα προστίθεται στην αρχή της συνδεδεμένης λίστας του στοιχείου *NTT[x]* του πίνακα μετάφρασης κόμβων. Για την ανάγνωση του κόμβου *x,* διαβάζεται η περιοχή προσωρινής αποθήκευσης ημερολογίου καθώς και όλοι οι τομείς της συνδεδεμένης λίστας του στοιχείου *NTT[x]*, για την συλλογή και συντακτική ανάλυση (parse) των καταχωρίσεων ημερολογίου του κόμβου *x,* πράξεις που οδηγούν με τη σειρά τους στην κατασκευή του λογικού κόμβου *x*.

4.5.1.1.3 Δομή τομέα, NTT και καταχωρίσεων ημερολογίου

Τα δεδομένα B^+- δένδρων(ST), προτού γραφτούν σε ένα τομέα της μνήμης φλας, περιβάλλονται από μία μικρή επικεφαλίδα η οποία περιέχει τα παρακάτω πεδία:

1. Checksum για τον έλεγχο λαθών κατά τη διάρκεια λειτουργιών ανάγνωσης.

2. NodeMode το οποίο καθορίζει την κατάσταση του κόμβου (δίσκου ή ημερολογίου).

Κάθε στοιχείο του πίνακα μετάφρασης κόμβων (NTT) περιέχει τα παρακάτω πεδία:

1. SectorList το οποίο δείχνει στους τομείς που περιέχουν τον αντίστοιχο κόμβο ή τις καταχωρίσεις ημερολογίου του κόμβου.

2. IsLeaf το οποίο λαμβάνει τιμή true εάν ο λογικός κόμβος είναι φύλλο.

3. LogVersion που είναι η πιο πρόσφατη έκδοση των καταχωρίσεων ημερολογίου του κόμβου.

Κάθε καταχώριση ημερολογίου (Log Entry) ενός κόμβου που βρίσκεται σε κατάσταση ημερολογίου, περιέχει τα παρακάτω πεδία:

1. NodeID το οποίο είναι το αναγνωριστικό του κόμβου με το οποίο ξεχωρίζουμε τις καταχωρίσεις ημερολογίου διαφορετικών κόμβων.

2. LogType: περιγράφει την λειτουργία που συντελείται στον λογικό κόμβο και μπορεί να είναι ένας από τους παρακάτω τύπους: ADD_KEY, DELETE_KEY και UPDATE_POINTER. Οι πρώτοι δύο τύποι χρησιμοποιούνται για την εισαγωγή και την διαγραφή αντίστοιχα, μίας καταχώρισης (κλειδί, δείκτης) σε έναν κόμβο του B^+- δένδρου. Ο τελευταίος τύπος είναι απαραίτητος για την ανανέωση του τελευταίου δείκτη ενός εσωτερικού κόμβου (internal node). Είναι σχετικά εύκολα να μετατρέψουμε έναν κόμβο σε καταχωρίσεις ημερολογίου, ή να κατασκευάσουμε έναν κόμβο από καταχωρίσεις ημερολογίου αυτών των τριών τύπων.

3. SequenceNumber: αυξάνεται κατά μία μονάδα κάθε φορά που παράγεται μία καταχώριση ημερολογίου για τον λογικό κόμβο. Χρησιμοποιείται για την χρήση των καταχωρίσεων ημερολογίου σύμφωνα με τη σειρά που παρήχθησαν.

4. LogVersion: πρόκειται για την πιο πρόσφατη έκδοση των καταχωρίσεων ημερολογίου.

Οι καταχωρίσεις ημερολογίου περιέχουν αρκετή πληροφορία, έτσι ώστε ακόμα και αν η εφαρμογή «καταρρεύσει» (crash) και χάσει τον πίνακα μετάφρασης κόμβων (που βρίσκεται στη RAM), τότε ο τελευταίος ανακατασκευάζεται σαρώνοντας ολόκληρη τη μνήμη φλας. Κάτι τέτοιο είναι αρκετά δαπανηρό, και μπορεί να αποφευχθεί ελέγχοντας περιοδικά τον πίνακα μετάφρασης κόμβων στη μνήμη φλας.

4.5.1.2 Βασικά ζητήματα αυτορύθμισης

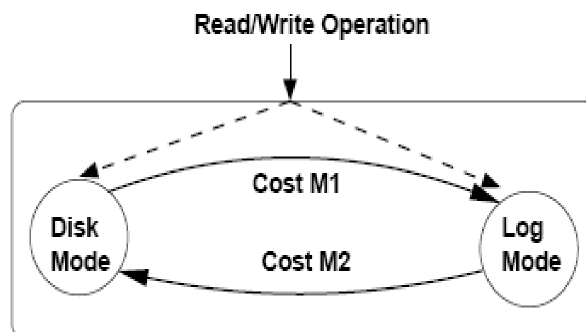
Για να γίνει ένα B^+- δένδρο(ST) αποδοτικό και αυτορθυμιζόμενο (self-tuning), η κατάσταση (mode) ενός κόμβου πρέπει να αποφασίζεται και να ανανεώνεται πολύ προσεκτικά. Επιπρόσθετα, το μέγεθος ενός κόμβου ευρετηρίου πρέπει να επιλέγεται με τον βέλτιστο τρόπο.

4.5.1.2.1 Αλγόριθμος εναλλαγής κατάστασης

Στην «καρδιά» των B^+- δένδρων(ST), υπάρχει ένας αλγόριθμος ο οποίος αποφασίζει πότε ένας κόμβος πρέπει να μεταβεί από κατάσταση δίσκου (Disk mode) σε κατάσταση ημερολογίου (Log mode). Η εναλλαγή μεταξύ των καταστάσεων εισάγει επιπλέον κόστος. Για την εναλλαγή ενός κόμβου *x* από κατάσταση ημερολογίου σε κατάσταση δίσκου, διαβάζονται πρώτα οι καταχωρίσεις ημερολογίου (log entries) του κόμβου, και στη συνέχεια ο κόμβος γράφεται σε κατάσταση δίσκου. Για την εναλλαγή του κόμβου *x* από κατάσταση δίσκου σε κατάσταση ημερολογίου, αρχικά ο κόμβος διαβάζεται σε κατάσταση δίσκου, κωδικοποιείται σε ένα σύνολο από καταχωρίσεις ημερολογίου που αντιπροσωπεύουν τον κόμβο, και τελικά οι καταχωρίσεις ημερολογίου τοποθετούνται στην περιοχή προσωρινής αποθήκευσης ημερολογίου (Log Buffer). Αντίστοιχα, τροποποιείται και το ανάλογο στοιχείο *NTT[x]* του πίνακα μετάφρασης κόμβων.

Στο σημείο αυτό θα παρουσιάσουμε τον αλγόριθμο εναλλαγής κατάστασης. Από τη στιγμή που ένας κόμβος σε κατάσταση δίσκου, είναι κατάλληλος για λειτουργίες ανάγνωσης και ένας κόμβος σε κατάσταση ημερολογίου, είναι κατάλληλος για λειτουργίες εγγραφής αντίστοιχα, τότε ένας κόμβος πρέπει να βρίσκεται σε κατάσταση δίσκου (ή σε κατάσταση ημερολογίου) όταν αναμένεται να υποστεί πολλές λειτουργίες ανάγνωσης (ή λειτουργίες εγγραφής). Από τη στιγμή που η εναλλαγή κατάστασης εισάγει επιπλέον κόστος, πρέπει να διασφαλιστεί ότι οι κόμβοι δεν αλλάζουν κατάσταση χωρίς λόγο. Επιπρόσθετα, επειδή οι αποφάσεις εναλλαγής κατάστασης πρέπει να λαμβάνονται δυναμικά, απαιτείται ένας **online** αλφόριθμος για εναλλαγή.

Μπορούμε να ανάγουμε το πρόβλημα εναλλαγής κατάστασης σε ένα σύστημα καθηκόντων δύο καταστάσεων (*Two-state Task System*), όπως φαίνεται και στο σχήμα 4.22 που ακολουθεί.



Σχήμα 4.22: Εναλλαγή κατάστασης ενός κόμβου B^+- δένδρου(ST).

Ένας κόμβος μπορεί να βρεθεί σε μία απο δύο καταστάσεις: δίσκου ή ημερολογίου. Μία ανάγνωση R ή μία εγγραφή W, μπορεί να εξυπηρετηθεί από έναν κόμβο σε όποια κατάσταση και αν είναι αυτός, αλλά τα κόστη είναι διαφορετικά ανάλογα με την κατάσταση του κόμβου. Ο κόμβος μπορεί να μεταβεί από τη μία κατάσταση στην άλλη με κάποιο κόστος. Ο βασικός στόχος είναι να χρησιμοποιηθεί ένας **online αλγόριθμος**, με βάση τον οποίο οι κόμβοι θα αλλάζουν δυναμικά την κατάστασή τους και θα ελαχιστοποιούν το συνολικό κόστος εξυπηρέτησης των αιτήσεων καθώς και το κόστος εναλλαγής.

Ο online αλγόριθμος εναλλαγής κατάστασης (SwitchMode Algorithm), παρουσιάζεται παρακάτω.

ALGORITHM **1.** : SWITCHMODE
(The following algorithm runs for each B-tree node n)

1. Initialize $S \leftarrow 0$ when migrate to the current mode.
2. For every read-write operation O
   - Suppose c_1 is the cost of serving O in current mode and c_2 is the cost of serving it in the other mode.
   - $S \leftarrow S + (c_1 - c_2)$
3. Suppose, M_1 and M_2 are the costs for transition between two modes. Then, switch to the other mode if $S \geq M_1 + M_2$

Ο παραπάνω αλγόριθμος είναι απλός και πρακτικός. Το μόνο που χρειάζεται είναι να διαμορφωθεί με βάση τα κόστη ανάγνωσης και εγγραφής σελίδας, ενώ υλοποιείται με έναν απλό μετρητή (*S*) ανά κόμβο. Ο αλγόριθμος «τρέχει» ανεξάρτητα σε κάθε κόμβο του δένδρου. Ο συγκεκριμένος αλγόριθμος είναι 3-ανταγωνιστικός (3-competitive) μιας και το κατώτερο όριο του ανταγωνιστικού λόγου (competitive ratio) είναι ίσο με 3. Ο αναταγωνιστικός λόγος ενός online αλγορίθμου είναι η χειρότερη περίπτωση του λόγου (ratio), μεταξύ του κόστους που οφείλεται στον online αλγόριθμο προς το κόστος που οφείλεται σε έναν offline βέλτιστο αλγόριθμο.

Όπως αναφέραμε και παραπάνω, ο αλγόριθμος εναλλαγής κατάστασης χρησιμοποιεί έναν μετρητή για κάθε κόμβο του δένδρου, ο οποίος αντιπροσωπεύει τη συσσωρευμένη (accumulated) διαφορά των κόστων των δύο καταστάσεων από την τελευταία εναλλαγή κατάστασης του κόμβου. Ο μετρητής του κόμβου *x* αποθηκεύεται στο στοιχείο *NTT[x]* του πίνακα μετάφρασης κόμβων. Υποθέτουμε ότι το κόστος ανάγνωσης σελίδας είναι c_r και το κόστος εγγραφής σελίδας είναι c_w. Επίσης υποθέτουμε ότι ένας κόμβος B^+- δένδρου(ST), σε κατάσταση δίσκου καταλαμβάνει k_s τομείς. Έτσι, ο υπολογισμός του κόστους σε κατάσταση δίσκου είναι απλός: κάθε λειτουργία ανάγνωσης και εγγραφής στον κόμβο κοστίζει $k_s * c_r$ και $k_s * c_w$, αντίστοιχα. Σε κατάσταση ημερολογίου, εάν μία λειτουργία εγγραφής παράγει l καταχωρίσεις ημερολογίου, και μία σελίδα της μνήμης φλας μπορεί να αποθηκεύσει μέχρι k_e καταχωρίσεις, τότε το κόστος της λειτουργίας εγγραφής είναι $l * c_w / k_e$. Παρόμοια, εάν οι καταχωρίσεις ημερολογίου ενός κόμβου είναι διασκορπισμένες σε p σελίδες της μνήμης φλας, τότε η ανάγνωση του κόμβου κοστίζει $p * c_r$. Εάν ο κόμβος είναι ήδη σε κατάσταση ημερολογίου, οι τιμές για τα p

και *l* αποφασίζονται με ακρίβεια. Από την άλλη πλευρά, εάν ο κόμβος είναι ήδη σε κατάσταση δίσκου, τότε γίνεται εκτίμηση των παραπάνω τιμών.

4.5.1.3 Βελτιστοποίηση B$^+$- δένδρων(ST)

4.5.1.3.1 Συμπίεση ημερολογίου και συλλογή απορριμμάτων

Κατά τη δημιουργία μίας δομής ευρετηρίου, ένας κόμβος B$^+$- δένδρου(ST) *x* μπορεί να ανανεωθεί αρκετές φορές, με αποτέλεσμα ένας μεγάλος αριθμός καταχωρίσεων ημερολογίου να διασκορπίζεται σε ένα μεγάλο αριθμό τομέων (sectors) της μνήμης φλας. Κάτι τέτοιο έχει δύο μειονεκτήματα. Πρώτον, το πεδίο NTT[x].SectorList γίνεται πολύ μεγάλο με αποτέλεσμα να αυξάνεται ο χώρος που καταλαμβάνεται από τον πίνακα μετάφρασης κόμβων (NTT). Δεύτερον, η ανάγνωση του κόμβου *x* γίνεται ακόμη πιο δαπανηρή μιας και διαβάζονται πιο πολύ τομείς. Για την αντιμετώπιση των παραπάνω προβλημάτων, χρησιμοποιούνται δύο τύποι **συμπίεσης ημερολογίου (log compaction)**.

Η πρώτη προσέγγιση είναι παρόμοια με αυτή που αναλύθηκε στην ενότητα 4.2.2.4. Ειδικότερα, όλες οι καταχωρίσεις ημερολογίου του κόμβου *x* διαβάζονται στη μνήμη RAM και στη συνέχεια ξαναγράφονται σε έναν μικρό αριθμό νέων τομέων. Κάτι τέτοιο μπορεί να βελτιώσει την απόδοση του συστήματος, μιας και οι καταχωρίσεις ημερολογίου του κόμβο *x* μπορεί να μοιράζονται κοινούς τομείς με καταχωρίσεις ημερολογίου άλλων κόμβων, εξοικονομώντας έτσι τομείς της μνήμης φλας. Παρ'όλα αυτά, η παραπάνω τεχνική δεν εγγυάται ένα άνω όριο στον αριθμό των τομέων που απαιτούνται για έναν κόμβο, μιας και ο αριθμός των καταχωρίσεων ημερολογίου ενός κόμβου μπορεί να μεγαλώσει απροσδόκητα με το πέρασμα του χρόνου.

Για την αντιμετώπιση της παραπάνω πρόκλησης, προτείνεται ένας **σημασιολογικός μηχανισμός συμπίεσης (semantic compaction mechanism)**, στον οποίο καταχωρίσεις ημερολογίου αντίθετης σημασιολογίας, απορρίπτονται κατά τη διάρκεια της συμπίεσης. Για παράδειγμα, εάν το δεδομένο *k* προστεθεί στον κόμβο *x* και στη συνέχεια αφαιρεθεί από αυτόν (εξαιτίας για παράδειγμα, μίας διάσπασης του κόμβου *x* όταν αυτός γεμίσει), τότε ο κόμβος *x* θα έχει καταχωρίσεις ημερολογίου ADD_KEY k και DELETE_KEY k. Οι δύο αυτές καταχωρίσεις ακυρώνουν η μία την άλλη με αποτέλεσμα να απορριφθούν. Παρομοίως, πολλαπλές καταχωρίσεις ημερολογίου UPDATE_POINTER για τον κόμβο *x* μπορούν να αντικατασταθούν από την τελευταία καταχώριση. Για τέτοιου είδους συμπίεση, λαμβάνεται υπόψιν ο αριθμός ακολουθίας (SequenceNumber) των καταχωρίσεων ημερολογίου. Εάν ένας κόμβος περιέχει το πολύ *n* αντικείμενα δεδομένων, θα έχει το πολύ *n+1* καταχωρίσεις ημερολογίου, φράσοντας το μέγεθος της συνδεδεμένης λίστας του στοιχείου *NTT[x]* στο *(n+1)/EntiresPerSector*. Ο σημασιολογικός μηχανισμός συμπίεσης απαιτεί κάθε καταχώριση ημερολογίου να έχει έναν αριθμό έκδοσης, ο οποίος αυξάνεται μετά από κάθε σημασιολογική συμπίεση. Μετα τη συμπίεση, το στοιχείο *NTT[x]* ανανεώνεται έτσι ώστε να περιλαμβάνει την τρέχουσα λίστα με τις διευθύνσεις των τομέων του κόμβου *x*.

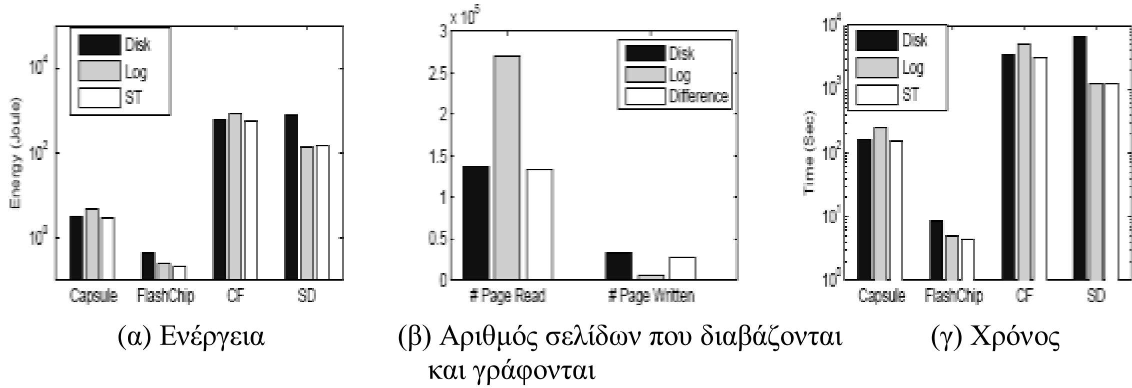
Ο σημασιολογικός μηχανισμός συμπίεσης εισάγει την έννοια των απαρχαιωμένων (obsolete) καταχωρίσεων ημερολογίου (αυτές που έχουν παλαιότερους αριθμούς έκδοσης). Για αυτόν το λόγο χρησιμοποιείται ένας **μηχανισμός συλλογής απορριμμάτων ημερολογίου (Log Garbage Collection-LGC)**, για την ανάκτηση χώρου. Ο μηχανισμός συλλογής απορριμμάτων ημερολογίου είναι διαφορετικός από το μηχανισμό συλλογής απορριμμάτων του διαχειριστή αποθήκευσης (Storage Manager). Ο τελευταίος ανακτά χώρο από «βρώμικες» σελίδες, ενώ ο μηχανισμός συλλογής απορριμμάτων ημερολογίου ανακτά χώρο από «βρώμικες» καταχωρίσεις ημερολογίου. Ο μηχανισμός συλλογής απορριμμάτων ημερολογίου ενεργοποιείται όταν η μνήμη φλας έχει λίγο διαθέσιμο αποθηκευτικό χώρο (όταν δηλαδή ο διαχειριστής αποθήκευσης αποτυγχάνει να δεσμεύσει έναν νέο τομέα). Ξεκινά σαρώνοντας ολόκληρη τη μνήμη φλας. Για κάθε τομέα, ο μηχανισμός συλλογής απορριμμάτων εξετάζει αρχικά την επικεφαλίδα του τομέα για να διαπιστώσει το κατά πόσο περιέχει καταχωρίσεις ημερολογίου. Αυτοί οι τομείς ονομάζονται **τομείς ημερολογίου (Log sectors)**. Σε κάθε τομέα ημερολογίου, ο μηχανισμός συλλογής απορριμμάτων μετράει τον αριθμό των απαρχαιωμένων καταχωρίσεων ημερολογίου. Εάν αυτός ο αριθμός ξεπερνά ένα συγκεκριμένο όριο, τότε ο συγκεκριμένος τομέας επιλέγεται για συλλογή απορριμμάτων. Στη συνέχεια, ο μηχανισμός συλλογής απορριμμάτων γράφει τις «φρέσκες» (fresh) καταχωρίσεις ημερολογίου στην περιοχή προσωρινής αποθήκευσης ημερολογίου (Log Buffer), αφαιρεί τις διευθύνσεις των τομέων από τον πίνακα μετάφρασης κόμβων, και επιστρέφει τον τομέα στον διαχειριστή αποθήκευσης. Τέλος, οι καταχωρίσεις ημερολογίου της περιοχής προσωρινής αποθήκευσης ημερολογίου (Log Buffer), μεταφέρονται στη μνήμη φλας, και οι νέες διευθύνσεις προστίθενται στον πίνακα μετάφρασης κόμβων.

4.5.1.3.2 Μεγαλύτερη περιοχή προσωρινής αποθήκευσης ημερολογίου

Εάν είναι εφικτό, συνίσταται η χρήση μεγαλύτερης περιοχής προσωρινής αποθήκευσης ημερολογίου (Log Buffer), μιας και η εγγραφή περισσότερων δεδομένων σε μία χρονική στιγμή έχει μικρότερο κόστος ανά byte, από την εγγραφή μικρότερης ποσότητας δεδομένων. Επιπρόσθετα, όταν γράφουμε στη μνήμη φλας, οι καταχωρίσεις ημερολογίου αναδιοργανώνονται με τέτοιο τρόπο ώστε, καταχωρίσεις του ίδιου κόμβου να βρίσκονται σε όσο το δυνατόν μικρότερο αριθμό τομέων. Κάτι τέτοιο μειώνει το χώρο που καταλαμβάνει ο πίνακας μετάφρασης κόμβων καθώς και το κόστος ανάγνωσης, μιας και απαιτούνται λιγότερες σελίδες να διαβαστούν για τη συλλογή όλων των καταχωρίσεων ημερολογίου ενός κόμβου.

4.5.2 Εκτίμηση της απόδοσης της FlashDB

Στην ενότητα αυτή θα παρουσιάσουμε τις επιδόσεις του B^+- δένδρου(ST), κάτω από διάφορους φόρτους εργασίας (workloads) και διαφορετικούς τύπους μνήμης φλας.

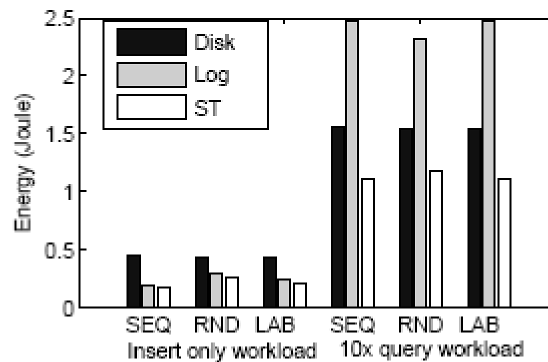


(α) Ενέργεια (β) Αριθμός σελίδων που διαβάζονται και γράφονται (γ) Χρόνος

Σχήμα 4.23: (α), **(**β**), (**γ**).** Ενέργεια και χρόνος με διαφορετικές συσκευές αποθήκευσης.

Από το σχήμα 4.23 (α) βλέπουμε ότι το B^+- δένδρο (Ημερολογίου) είναι 40% και 80% πιο αποδοτικό από το B^+- δένδρο (Δίσκου), όταν χρησιμοποιείται ως μνήμη φλας το FlashChip της Samsung και το Secure Digital της Kingston, αντίστοιχα. Από την άλλη πλευρά, το B^+- δένδρο (Δίσκου) είναι κατά 38% και 32% πιο αποδοτικό από το B^+- δένδρο (Ημερολογίου) όταν χρησιμοποιείται ως μνήμη φλας το Capsule της Toshiba και το Compact flash της Sandisk, αντίστοιχα.

Το B^+- δένδρο (ST) αποδίδει καλύτερα ή το ίδιο καλά με το καλύτερο εκ των B^+- δένδρο (Δίσκου) και B^+- δένδρο (Ημερολογίου). Αυτό οφείλεται στο γεγονός ότι στο B^+- δένδρο (ST), κάθε κόμβος ξεχωριστά παραμένει στη βέλτιστη κατάσταση ανάλογα με τον αντίστοιχο φόρτο εργασίας και τον τύπο μνήμης φλας που χρησιμοποιείται. Το B^+- δένδρο (ST) καταναλώνει λιγότερη ενέργεια από το B^+- δένδρο (Ημερολογίου), γιατί μερικοί κόμβοι κοντά στον κόμβο ρίζα, παραμένουν σε κατάσταση δίσκου.



Σχήμα 4.24: Ενέργεια και χρόνος για διαφορετικούς φόρτους εργασίας.

Με βάση το σχήμα 4.24, το B^+- δένδρο (ST) είναι πολύ πιο αποδοτικό σε σχέση με το B^+- δένδρο (Δίσκου) και το B^+- δένδρο (Ημερολογίου), υπό διαφορετικούς φόρτους εργασίας. Με SEQ συμβολίζουμε τον φόρτο εργασίας που αποτελείται από 30000 στοιχεία σε αύξουσα σειρά, με RND συμβολίζουμε τον φόρτο εργασίας που αποτελείται από 30000 τυχαία στοιχεία και με LAB συμβολίζουμε τον φόρτο

εργασίας που αποτελείται από δεδομένα θερμοκρασίας, τα οποία συλλέχθησαν από 35 αισθητήρες.

Βιβλιογραφία κεφαλαίου

[1] Chin-Hsien Wu, Tei-Wei Kuo and Li Ping Chang. An Efficient B-Tree Layer Implementation for Flash-Memory Storage Systems. (χρήση στις σελίδες 47-64)

[2] Dongwon Kang, Dawoon Jung, Jeong-Uk Kang and Jin-Soo Kim. μ - Tree: An Ordered Index Structure for NAND Flash Memory. (χρήση στις σελίδες 64-78)

[3] Suman Nath and Aman Kansal. FlashDB: Dynamic Self-tuning Database for NAND Flash. (χρήση στις σελίδες 78-87)

Κεφάλαιο 5

Δομές δεδομένων για γεωγραφικά δεδομένα

Περιεχόμενα

5.1 Εισαγωγή……………………………………………………….......... 90

5.2 R-δένδρα.. 90

5.2.1 Βασικές πράξεις σε R-δένδρα... **93**

5.2.1.1 Αναζήτηση στοιχείου.. 93

5.2.1.2 Εισαγωγή στοιχείου... 93

5.2.1.3 Διαγραφή στοιχείου... 95

5.2.1.4 Ανανέωση στοιχείου... 96

5.2.1.5 Διάσπαση κόμβου.. 96

5.3 Αποδοτική υλοποίηση R-δένδρων σε μνήμη φλας............................ 98

5.3.1 Λόγοι δημιουργίας της υλοποίησης.. 98

5.3.2 Επισκόπηση της υλοποίησης...100

5.3.3 Δομές δεδομένων...101

5.3.4 Ανάλυση πολυπλοκότητας..104

Βιβλιογραφία κεφαλαίου...105

5.1 Εισαγωγή

Η μνήμη φλας έχει πλέον καθιερωθεί ως η βασική αποθηκευτική συσκευή σε πολλές εφαρμογές. Η επανάσταση που σημειώνεται στα δίκτυα κινητής τηλεφωνίας, έχει οδηγήσει στην δημιουργία νέων εφαρμογών, κατάλληλες για συσκευές παλάμης (hand-held devices). Μία τέτοια εφαρμογή είναι το **σύστημα γεωγραφικής πληροφόρησης (Geographic Information System-GIS)**. Μία βασική τεχνική πρόκληση, σχετική με τον σχεδιασμό του αποθηκευτικού συστήματος της παραπάνω εφαρμογής, είναι το πως θα προσπελαύνονται αποδοτικά οι διάφορες γεωγραφικές πληροφορίες, όπως για παράδειγμα ηλεκτρονικοί χάρτες, και πως ακριβώς θα αποθηκεύονται αυτές οι πληροφορίες στις συσκευές παλάμης. Η υποστήριξη τέτοιου είδους εφαρμογών, με ανάγκες δομών ευρετηρίου κατάλληλες για γεωγαφικά δεδομένα (spatial index structures), είναι πολλή σημαντική για την απόδοση των ενσωματωμένων συστημάτων (embedded systems), ειδικά από τη στιγμή που η χωρητικότητα της μνήμης φλας αυξάνεται με ταχύτατους ρυθμούς τα τελευταία χρόνια.

Στις ενότητες που ακολουθούν, θα παρουσιαστόυν αρχικά τα **R-δένδρα**, τα οποία είναι κατάλληλα για γεωγραφικά δεδομένα και υλοποιούνται συνήθως σε σκληρούς δίσκους. Στη συνέχεια του κεφαλαίου, θα μελετήσουμε μία **αποδοτική υλοποίηση των R-δένδρων, κατάλληλη για χρήση σε μνήμη φλας**.

5.2 R-δένδρα

Ένα **R-δένδρο** είναι ένα ισορροπημένο (balanced) δένδρο παρόμοιο με το B-δένδρο. Οι καταχωρίσεις ευρετηρίου (index records), αποθηκεύονται στους κόμβους φύλλα, και περιέχουν δείκτες προς τα αντικείμενα δεδομένων (data objects). Εάν η δομή ευρετηρίου αποθηκεύεται σε σκληρό δίσκο, τότε οι κόμβοι του R-δένδρου βρίσκονται στις σελίδες του δίσκου και γενικά η δομή είναι έτσι σχεδιασμένη ώστε η αναζήτηση ενός γεωγραφικού δεδομένου να απαιτεί την επίσκεψη ενός μικρού αριθμού κόμβων. Το R-δένδρο είναι μία δυναμική δομή ευρετηρίου, όπου οι πράξεις εισαγωγής και διαγραφής στοιχείων μπορούν να «αναμιχθούν» (intermixed) με πράξεις αναζητήσεων, χωρίς να απαιτείται περιοδική αναδιοργάνωση του δένδρου.

Μία βάση δεδομένων με γεωγραφικά δεδομένα, αποτελείται από μία συλλογή πλειάδων (tuples) οι οποίες αντιπροσωπεύουν γεωγραφικά αντικείμενα. Κάθε πλειάδα της βάσης δεδομένων έχει ένα μοναδικό αναγνωριστικό το οποίο χρησιμοποιείται κατά την ανάκτησή της. Οι κόμβοι φύλλα του R-δένδρου , περιέχουν καταχωρίσεις ευρετηρίου της μορφής:

(I, tuple-identifier)

όπου το *tuple-identifier* είναι το αναγνωριστικό μίας πλειάδας της βάσης δεδομένων, ενώ με I συμβολίζεται ένα παραλληλόγραμμο διάστασης n που επικαλύπτει τα γεωγραφικά δεδομένα της συγκεκριμένης πλειάδας.

$$I = (I_{0,} I_{1,} ..., I_{n-1})$$

Εδώ το *n* είναι ο αριθμός των διαστάσεων, και το I_i είναι ένα κλειστό διάστημα [a, b] το οποίο περιγράφει την έκταση του αντικειμένου κατά μήκος της διάστασης *i*.

Οι εσωτερικοί κόμβοι (non-leaf nodes) του R-δένδρου περιέχουν καταχωρίσεις της μορφής:

(I, child-pointer)

όπου το *child-pointer* είναι η διεύθυνση ενός κόμβου χαμηλότερου επιπέδου στο R-δένδρο, ενώ το παραλληλόγραμμο *I* περιέχει τα παραλληλόγραμμα όλων των χαμηλότερων κόμβων.

Έστω *M* ο μέγιστος αριθμός καταχωρίσεων που αποθηκεύονται σε ένα κόμβο του R-δένδρου, και $m \leq \frac{M}{2}$ μία παράμετρος που καθορίζει τον ελάχιστο αριθμό καταχωρίσεων σε έναν κόμβο. Ένα R-δένδρο, έχει τις παρακάτω ιδιότητες:

1. Κάθε κόμβος φύλλο περιέχει μεταξύ *m* και *M* καταχωρίσεις ευρετηρίου, εκτός αν είναι κόμβος ρίζα (root node).

2. Σε κάθε καταχώριση ευρετηρίου *(I, tuple-identifier)* ενός κόμβου φύλλου, το *I* είναι το ελάχιστο παραλληλόγραμμο που καλύπτει χωρικά, τα αντικείμενα δεδομένων διάστασης n, που ανήκουν στην πλειάδα με αναγνωριστικό *tuple-identifier.*

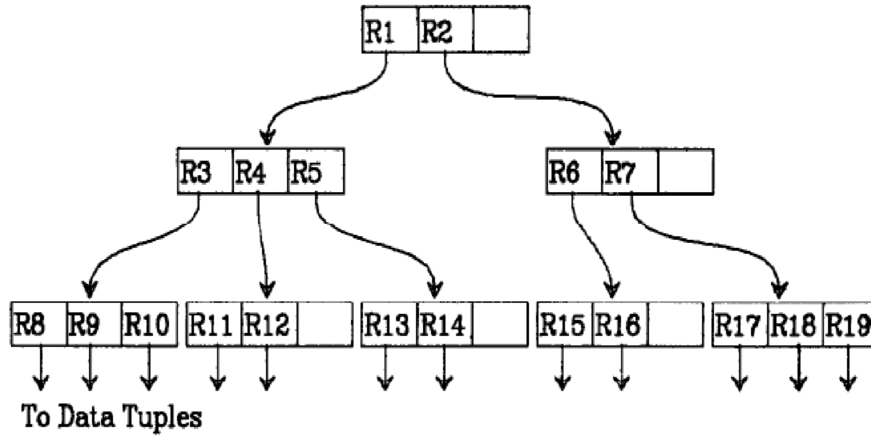
3. Κάθε εσωτερικός κόμβος (non-leaf node) έχει μεταξύ *m* και *M* κόμβους παιδιά, εκτός αν είναι κόμβος ρίζα (root node).

4. Σε κάθε καταχώριση *(I, child-pointer)* ενός εσωτερικού κόμβου, το *I* είναι το ελάχιστο παραλληλόγραμμο που περιέχει τα παραλληλόγραμμα του κόμβου παιδίου του.

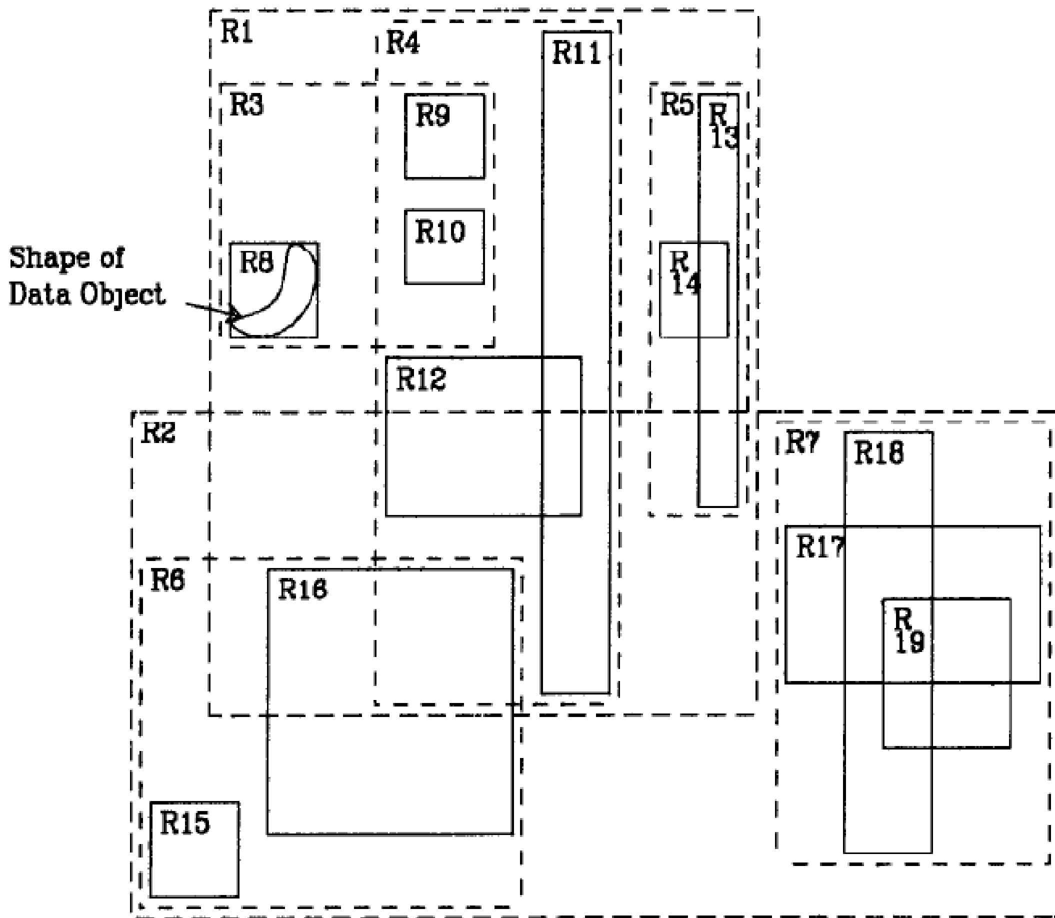
5. Ο κόμβος ρίζα έχει τουλάχιστον δύο παιδιά, εκτός αν είναι κόμβος φύλλο.

6. Όλοι οι κόμβοι φύλλα βρίσκονται στο ίδιο επίπεδο στο R-δένδρο.

Τα σχήματα 5.1 και 5.2 που ακολουθούν, παρουσιάζουν την δομή ενός R-δένδρου καθώς και τον τρόπο με τον οποίο ομαδοποιούνται τα γεωγραφικά δεδομένα της βάσης σε παραλληλόγραμμα. Αξίζει να παρατηρήσουμε ότι το παραλληλόγραμμο R1 περιέχει όλα τα παραλληλόγραμμα του αριστερού υποδένδρου (R3, R4, R5, R8, R9, R10, R11, R12, R13, R14), ως αποτέλεσμα της ιδιότητας 4 που αναφέραμε παραπάνω. Το ίδιο ισχύει και για το R2, με το δεξί υποδένδρο.



Σχήμα 5.1: Δομή ενός R-δένδρου.



Σχήμα 5.2: Ομαδοποίηση δεδομένων σε παραλληλόγραμμα.

Το ύψος ενός R-δένδρου που περιέχει N καταχωρίσεις ευρετηρίου είναι τουλάχιστον ίσο με $|\log_m N| - 1$. Ο μέγιστος αριθμός κόμβων είναι $\left|\frac{N}{m}\right| + \left|\frac{N}{m^2}\right| + ... + 1$.

5.2.1 Βασικές πράξεις σε R-δένδρα

Στην ενότητα αυτή θα παρουσιάσουμε τις βασικές πράξεις αναζήτησης, εισαγωγής, διαγραφής και ανανέωσης στοιχείου σε ένα R-δένδρο.

5.2.1.1 Αναζήτηση στοιχείου

Ο αλγόριθμος αναζήτησης εξερευνεί το R-δένδρο από τη ρίζα προς τα φύλλα, με τρόπο παρόμοιο με αυτόν των B-δένδρων. Παρ'όλα αυτά, κατά την επίσκεψη ενός κόμβου, μπορεί να χρειαστεί να εξερευνηθεί παραπάνω από ένα υποδένδρο, με αποτέλεσμα να μην παρέχονται εγγυήσεις καλής απόδοσης χειρότερης περίπτωσης (good worst-case performance). Όμως, με τα περισσότερα είδη δεδομένων, ο αλγόριθμος ανανέωσης διατηρεί το R-δένδρο σε μία κατάσταση που επιτρέπει στον αλγόριθμο αναζήτησης να εξετάζει δεδομένα που βρίσκονται κοντά στην περιοχή αναζήτησης.

Από το σημείο αυτό και μετά, θα συμβολίζουμε με *EI* το παραλληλόγραμμο *I* μίας καταχώρισης ευρετηρίου *E,* και με E_p το *tuple-identifier* ή το *child-pointer,* ανάλογα με το αν ο κόμβος είναι φύλλο ή εσωτερικός αντίστοιχα.

Αλγόριθμος **αναζήτησης**. Δοθέντος ενός R-δένδρου με ρίζα ***T*,** να βρεθούν όλες οι καταχωρίσεις ευρετηρίου των οποίων τα παραλληλόγραμμα περιέχουν/επικαλύπτουν (overlap) το προς αναζήτηση παραλληλόγραμμο *S*.

S1 [Εξερεύνηση υποδένδρων]: Εάν ο κόμβος *T* δεν είναι φύλλο, έλεγχεται κάθε καταχώριση *E* του κόμβου για να αποφασιστεί το κατά πόσο το παραλληλόγραμμο *EI* της καταχώρισης, περιέχει το *S*. Για όσες καταχωρίσεις ισχύει αυτό, εφαρμόζεται ο αλγόριθμος αναζήτησης στους κόμβους που δείχνουν οι δείκτες E_p των καταχωρίσεων.

S2 [Εξερεύνηση κόμβου φύλλου]: Εάν ο κόμβος *T* είναι φύλλο, ελέγχεται κάθε καταχώριση *E* του κόμβου για να αποφασιστεί το κατά πόσο το παραλληλόγραμμο *EI* της καταχώρισης, περιέχει το *S*. Εάν κάτι τέτοιο ισχύει, η καταχώριση *E* είναι μία κατάλληλη καταχώριση (qualifying record).

5.2.1.2 Εισαγωγή στοιχείου

Η εισαγωγή μίας καταχώρισης ευρετηρίου σε ένα R-δένδρο, είναι παρόμοια με την εισαγωγή στοιχείων σε ένα B-δένδρο, με την έννοια ότι οι καινούργιες καταχωρίσεις ευρετηρίου αποθηκεύονται στους κόμβους φύλλα, οι κόμβοι που υπερχειλίζουν (overflow) διασπόνται (split), και οι διασπάσεις μεταδίδονται προς τα ανώτερα επίπεδα του δένδρου.

Αλγόριθμος **εισαγωγής**. Εισαγωγή μίας καινούργιας καταχώρισης ευρετηρίου *E* σε ένα R-δένδρο.

I1 [Εύρεση θέσης για την καινούργια καταχώριση]: Εφαρμογή του αλγορίθμου **ChooseLeaf** για την επιλογή ενός κόμβου φύλλου *L* στο οποίο θα αποθηκευτεί η καταχώριση *E*.

I2 [Προσθήκη καταχώρισης στον κόμβο φύλλο]: Εάν ο κόμβος φύλλο *L* έχει αρκετό χώρο για νέες καταχωρίσεις, τότε η καταχώριση *E* αποθηκεύεται στο φύλλο. Σε διαφορετική περίπτωση, εφαρμόζεται ο αλγόριθμος **SplitNode** που διασπά τον κόμβο *L* σε *L* και *LL*, οι οποίοι περιέχουν την καταχώριση *E* και όλες τις παλαιές καταχωρίσεις του κόμβου *L*.

I3 [Μετάδοση αλλαγών προς τα επάνω επίπεδα του R-δένδρου]: Εφαρμογή του αλγορίθμου **AdjustTree** στον κόμβο φύλλο *L*, καθώς και στον *LL* εάν προηγουμένως συναίβει διάσπαση του κόμβου *L*.

I4 [Αύξηση ύψους του R-δένδρου]: Εάν η διαδικασία διάσπασης κόμβων φτάσει στον κόμβο ρίζα, δημιουργείται νέος κόμβος ρίζα με παιδιά τους δύο νεοσύστατους κόμβους.

Αλγόριθμος **ChooseLeaf**. Επιλογή ενός κόμβου φύλλου για την τοποθέτηση της καινούργιας καταχώρισης *E*.

CL1 [Αρχικοποίηση]: Θέτουμε ως *N* τον κόμβο ρίζα του R-δένδρου.

CL2 [Έλεγχος φύλλου]: Εάν ο *N* είναι φύλλο, τότε επιστρέφεται ο *N*.

CL3 [Επιλογή υποδένδρου]: Εάν ο *N* δεν είναι φύλλο, τότε έστω *F* η καταχώριση του κόμβου *N*, της οποίας το παραλληλόγραμμο *FI* χρειάζεται την ελάχιστη διεύρυνση έτσι ώστε να συμπεριλάβει το παραλληλόγραμμο *EI*.

CL4 [Κάθοδος μέχρι να συναντηθεί ένας κόμβος φύλλο]: Θέτουμε ως *N* τον κόμβο παιδί που υποδεικνύει ο δείκτης F_p, και επιστρέφουμε ξανά στο βήμα CL2.

Αλγόριθμος **AdjustTree**. Ανάβαση από έναν κόμβο φύλλο *L* προς τη ρίζα, ρυθμίζοντας τα επικαλυπτόμενα παραλληλόγραμμα (covering rectangles) και μεταδίδοντας τις διασπάσεις κόμβων όπου είναι απαραίτητο.

AT1 [Αρχικοποίηση]: Θέτουμε *N*=*L*. Εάν ο κόμβος *L* είχε διασπαστεί προηγουμένως, τότε θέτουμε ως *NN* τον νέο κόμβο που είναι αποτέλεσμα της διάσπασης.

AT2 [Έλεγχος για τερματισμό]: Εάν ο *N* είναι ρίζα, τότε τέλος.

AT3 [Ρύθμιση επικαλυπτόμενου παραλληλογράμμου στην καταχώριση πατέρα]: Έστω *P* ο κόμβος πατέρα του κόμβου *N*, και έστω E_N η καταχώριση του κόμβου *P* που δείχνει προς τον κόμβο *N*. Ακολουθεί ρύθμιση του παραλληλογράμμου $E_N I$ έτσι ώστε να περιέχει «σφιχτά» όλα τα παραλληλόγραμμα των καταχωρίσεων του κόμβου *N*.

AT4 [Μετάδοση διάσπασης κόμβων προς τα επάνω]: Εάν ο κόμβος *N* έχει έναν αδελφό *NN* ως αποτέλεσμα προηγούμενης διάσπασης, τότε δημιουργείται μία νέα

καταχώριση E_{NN} με τον δείκτη $E_{NN}p$ να δείχνει προς τον κόμβο *NN*, και το παραλληλόγραμμο $E_{NN}I$ να περιλαμβάνει όλα τα παραλληλόγραμμα του κόμβου *NN*. Η καταχώριση E_{NN}, προστίθεται στον κόμβο *P,* εάν ο τελευταίος διαθέτει ελεύθερο χώρο. Διαφορετικά, εφαρμόζεται ο αλγόριθμος **SplitNode**, για τη δημιουργία των κόμβων *P* και *PP*, οι οποίοι περιέχουν την καταχώριση E_{NN} και όλες τις παλαιές καταχωρίσεις του *P*.

AT5 [Μεταφορά προς το επόμενο επίπεδο]: Θέτουμε *N=P* και *NN=PP* εάν έχει πραγματοποιηθεί διάσπαση. Επιστροφή στο βήμα AT2.

Ο αλγόριθμος **SplitNode**, παρουσιάζεται στην ενότητα 5.2.1.5

5.2.1.3 Διαγραφή στοιχείου

Αλγόριθμος **διαγραφής**. Διαγραφή μίας καταχώρισης ευρετηρίου *E* από ένα R-δένδρο.

D1 [Εύρεση κόμβου που περιέχει την καταχώριση *E*]: Εφαρμογή του αλγορίθμου **FindLeaf**, για τον εντοπισμό του κόμβου φύλλου *L*, που περιέχει την καταχώριση *E*. Η διαδικασία τερματίζεται αν η καταχώριση δεν εντοπιστεί.

D2 [Διαγραφή καταχώρισης]: Διαγραφή της καταχώρισης *E* από τον κόμβο φύλλο *L*.

D3 [Μετάδοση αλλαγών]: Εφαρμογή του αλγορίθμου **CondenseTree**, στον κόμβο φύλλο *L*.

D4 [Μείωση του ύψους του δένδρου]: Εάν ο κόμβος ρίζα έχει μόνο ένα παιδί, αμέσως μετά την ρύθμιση (adjust) του δένδρου, θέτουμε το παιδί ως τον νέο κόμβο ρίζα.

Αλγόριθμος **FindLeaf**. Δοθέντος ενός R-δένδρου με ρίζα *T,* να βρεθεί ο κοόμβος φύλλο ο οποίος περιέχει την καταχώριση ευρετηρίου *E*.

FL1 [Εξερεύνηση υποδένδρων]: Εάν ο κόμβος *T* δεν είναι φύλλο, τότε εξετάζεται κάθε καταχώριση *F* του κόμβου *T*, για να διαπιστωθεί το κατά πόσο το παραλληλόγραμμο *FI* επικαλύπτει/περιέχει το *EI*. Για κάθε καταχώριση που ικανοποιεί την παραπάνω συνθήκη, εφαρμόζεται ο αλγόριθμος **FindLeaf** στο δένδρο με ρίζα τον κόμβο που υποδεικνύει ο δείκτης F_p της καταχώρισης. Η διαδικασία συνεχίζεται μέχρι να βρεθεί η καταχώριση ευρετηρίου *E* ή μέχρι να εξεταστούν όλες οι καταχωρίσεις.

FL2 [Εξέταση κόμβου φύλλου για το αν περιέχει την επιθυμητή καταχώριση]: Εάν ο κόμβος *T* είναι φύλλο, τότε εξετάζεται κάθε καταχώρισή του για να διαπιστωθεί το αν περιέχει την καταχώριση ευρετηρίου *E*. Εάν κάτι τέτοιο ισχύει, τότε επιστρέφεται ο κόμβος *T*.

Αλγόριθμος **CondenseTree**. Δοθέντος ενός κόμβου φύλλου L από τον οποίο διαγράφηκε μία καταχώριση, τότε ο κόμβος απαλείφεται (eliminate) εάν περιέχει πολύ λίγες καταχωρίσεις. Οι εναπομείναντες καταχωρίσεις του κόμβου επανατοποθετούνται σε άλλους κόμβους. Η διαδικασία απαλειφής κόμβων, μεταδίδεται προς τα επάνω επίπεδα του R-δένδρου, αν βέβαια χρειαστεί. Επιπλέον, ρυθμίζονται και τα παραλληλόγραμμα των κόμβων στο μονοπάτι προς τη ρίζα, κάνοντάς τα όσο το δυνατόν μικρότερα.

CT1 [Αρχικοποίηση]: Θέτουμε $N=L$. Έστω Q το σύνολο των κόμβων που έχουν απαλειφθεί, το οποίο αρχικά είναι άδειο.

CT2 [Εύρεση καταχώρισης πατέρα]: Εάν ο N είναι κόμβος ρίζα, τότε ο αλγόριθμος μεταβαίνει στο βήμα CT6. Διαφορετικά, υποθέτουμε ότι ο κόμβος πατέρας του N είναι ο P. Έστω E_N η καταχώριση του κόμβου P που δείχνει προς τον κόμβο N.

CT3 [Απαλειφή κόμβου N]: Εάν ο κόμβος N έχει λιγότερες από m καταχωρίσεις, τότε διαγράφεται η καταχώριση E_N από τον κόμβο P και ο κόμβος N προστίθεται στο σύνολο Q.

CT4 [Ρύθμιση παραλληλογράμμων]: Εάν ο κόμβος N δεν απαλειφθεί, τότε ρυθμίζεται το παραλληλόγραμμο $E_N I$ ώστε να περιέχει «σφιχτά» όλες τις καταχωρίσεις του κόμβου N.

CT5 [Μεταφορά στο αμέσως επόμενο επίπεδο του δένδρου]: Θέτουμε $N=P$ και η διαδικασία επαναλαμβάνεται από το βήμα CT2.

CT6 [Επανατοποθέτηση ορφανών καταχωρίσεων]: Επανατοποθέτηση όλων των καταχωρίσεων των κόμβων του συνόλου Q. Οι καταχωρίσεις, των κόμβων **φύλλων** που έχουν απαλειφθεί, επανατοποθετούνται σε κόμβους φύλλα με βάση τον αλγόριθμο **εισαγωγής**. Οι καταχωρίσεις κόμβων που δεν είναι φύλλα, τοποθετούνται σε υψηλότερες θέσεις στο δένδρο.

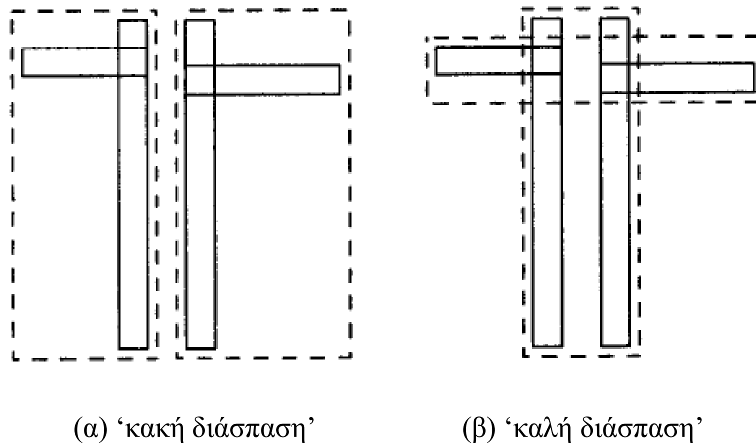
5.2.1.4 Ανανέωση στοιχείου

Εάν μία πλειάδα δεδομένων (data tuple) ανανεωθεί, έτσι ώστε το παραλληλόγραμμο που την περιέχει να τροποποιηθεί, τότε η αντίστοιχη καταχώριση ευρετηρίου στο R-δένδρο, πρέπει να διαγραφεί, να ανανεωθεί, και στη συνέχεια να επανατοποθετηθεί στη σωστή θέση μέσα στο δένδρο.

5.2.1.5 Διάσπαση κόμβου

Για να τοποθετήσουμε μία καινούργια καταχώριση σε ένα γεμάτο κόμβο που περιέχει M καταχωρίσεις, είναι απαραίτητο να διαμερίσουμε τις $M+1$ καταχωρίσεις σε δύο κόμβους. Η διάσπαση πρέπει να γίνει με τέτοιο τρόπο ώστε να μην χρειάζεται να εξεταστούν και οι δύο κόμβοι σε μεταγενέστερες αναζητήσεις. Από τη στιγμή που η απόφαση για το κατά πόσο πρέπει να επισκεφτούμε ένα κόμβο, εξαρτάται από το αν

το παραλληλόγραμμο του κόμβου περιέχει την υπό αναζήτηση περιοχή (search area), η συνολική περιοχή που καταλαμβάνουν τα παραλληλόγραμμα των δύο κόμβων, μετά από τη διάσπαση, πρέπει να είναι η ελάχιστη δυνατή. Στο σχήμα 5.3 που ακολουθεί, η περιοχή που καταλαμβάνουν τα παραλληλόγραμμα στην περίπτωση 'κακής διάσπασης' είναι αρκετά μεγαλύτερη από αυτή που καταλαμβάνεται στην περίπτωση 'καλής διάσπασης'.



(α) 'κακή διάσπαση' (β) 'καλή διάσπαση'

Σχήμα 5.3: (α), (β). Στιγμιότυπα παραλληλογράμμων μετά από τη διάσπαση ενός κόμβου.

Θα παρουσιάσουμε τώρα αλγορίθμους για τη διαμέριση του συνόλου των $M+1$ καταχωρίσεων σε δύο ομάδες (groups), μία για κάθε κόμβο.

Εξαντλητικός αλγόριθμος (Exhaustive algorithm).

Ο πιο ξεκάθαρος τρόπος για την εύρεση της ελάχιστης περιοχής που καταλαμβάνουν τα παραλληλόγραμμα μετά από τη διάσπαση, είναι να παράγουμε όλες τις δυνατές ομαδοποιήσεις (groupings) και να επιλέξουμε την καλύτερη. Παρ'όλα αυτά, όλες οι δυνατές ομαδοποιήσεις είναι προσεγγιστικά ίσες με 2^{M-1}, και μία εύλογη τιμή του M είναι ίση με 50, με αποτέλεσμα το πλήθος των δυνατών διασπάσεων να είναι πολύ μεγάλο.

Αλγόριθμος τετραγωνικής διάσπασης (Quadratic split algorithm).

Διαμέριση ενός συνόλου $M+1$ καταχωρίσεων ευρετηρίου σε δύο ομάδες.

QS1 [Επιλογή πρώτης καταχώρισης για κάθε ομάδα]: Εφαρμογή του αλγορίθμου **PickSeeds** για την επιλογή των δύο πρώτων καταχωρίσεων, μία για κάθε ομάδα.

QS2 [Έλεγχος τερματισμού]: Εάν έχουν τοποθετηθεί στις ομάδες όλες οι καταχωρίσεις, τότε τέλος. Εάν μία ομάδα έχει πολύ λίγες καταχωρίσεις, έτσι ώστε όλες οι υπόλοιπες καταχωρίσεις να πρέπει να τοποθετηθούν σε αυτή για να φτάσει τον ελάχιστο αριθμό m, τότε τοποθετούνται σε αυτή και η διαδικασία τερματίζει.

QS3 [Επιλογή καταχώρισης για ανάθεση]: Εφαρμογή του αλγορίθμου **PickNext** για την επιλογή της επόμενης προς ανάθεση καταχώρισης. Η τοποθέτησή της γίνεται στην ομάδα της οποίας το παραλληλόγραμμο θα χρειαστεί την ελάχιστη δυνατή μεγένθυνση έτσι ώστε να την συμπεριλάβει. Επιστροφή στο βήμα QS2.

Αλγόριθμος **PickSeeds**. Επιλογή δύο καταχωρίσεων οι οποίες θα είναι τα πρώτα στοιχεία των δύο ομάδων.

PS1: Για κάθε ζευγάρι καταχωρίσεων E_1 και E_2, δημιουργείται ένα παραλληλόγραμμο J το οποίο περιέχει τα παραλληλόγραμμα $E_1 I$ και $E_2 I$. Στη συνέχεια υπολογίζεται η τιμή $d = area(J) - area(E_1 I) - area(E_2 I)$

PS2: Επιλογή του ζευγαριού με τη μεγαλύτερη τιμή *d*.

Αλγόριθμος **PickNext**. Επιλογή της επόμενης καταχώρισης για ομαδοποίηση.

PN1 [Υπολογισμός κόστους τοποθέτησης καταχώρισης και στις δύο ομάδες]: Για κάθε καταχώριση *E* που δεν έχει τοποθετηθεί σε ομάδα, υπολογίζεται η τιμή d_1 = ποσό αύξησης περιοχής που απαιτείται στο παραλληλόγραμμο της ομάδας 1 έτσι ώστε να περιλαμβάνει και το παραλληλόγραμμο *EI*. Υπολογισμός τιμής d_2, με παρόμοιο τρόπο για την ομάδα 2.

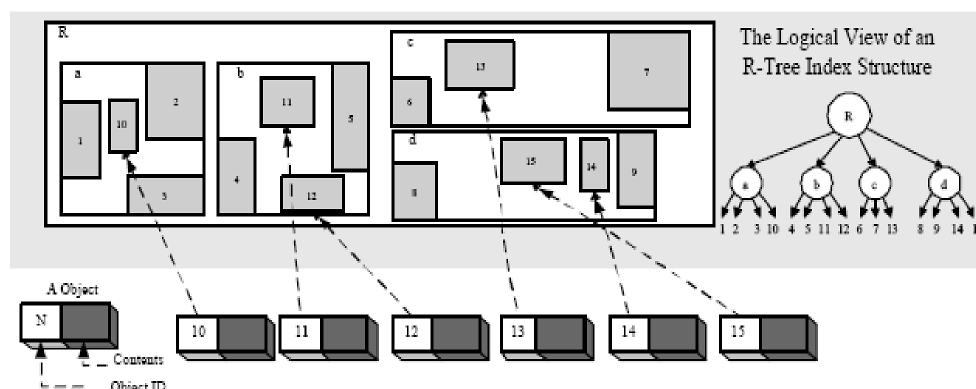
PN2 [Επιλογή καταχώρισης]: Επιλογή καταχώρισης με τη μέγιστη διαφορά d_1 - d_2.

5.3 Αποδοτική υλοποίηση R-δένδρων σε μνήμη φλας

Στις ενότητες που ακολουθούν, θα παρουσιαστεί μία αποδοτική υλοποίηση των R-δένδρων, η οποία είναι κατάλληλη προς χρήση σε συστήματα με μνήμη φλας. Η υλοποίηση γίνεται απευθείας πάνω από το FTL (Flash Translation Layer), χωρίς καμία μεταβολή στα ήδη υπάρχοντα συστήματα εφαρμογών. Πιο συγκεκριμένα, θα μελετηθούν οι λόγοι που οδήγησαν στη δημιουργία αυτής της υλοποίησης, οι δομές δεδομένων της υλοποίησης, η διαχείρησή τους καθώς και η ανάλυση πολυπλοκότητας της υλοποίησης.

5.3.1 Λόγοι δημιουργίας της υλοποίησης

Υποθέτουμε ότι πρέπει να εισάγουμε έξι γεωγραφικά/χωρικά αντικείμενα (spatial oblects), στο R-δένδρο του σχήματος 5.4



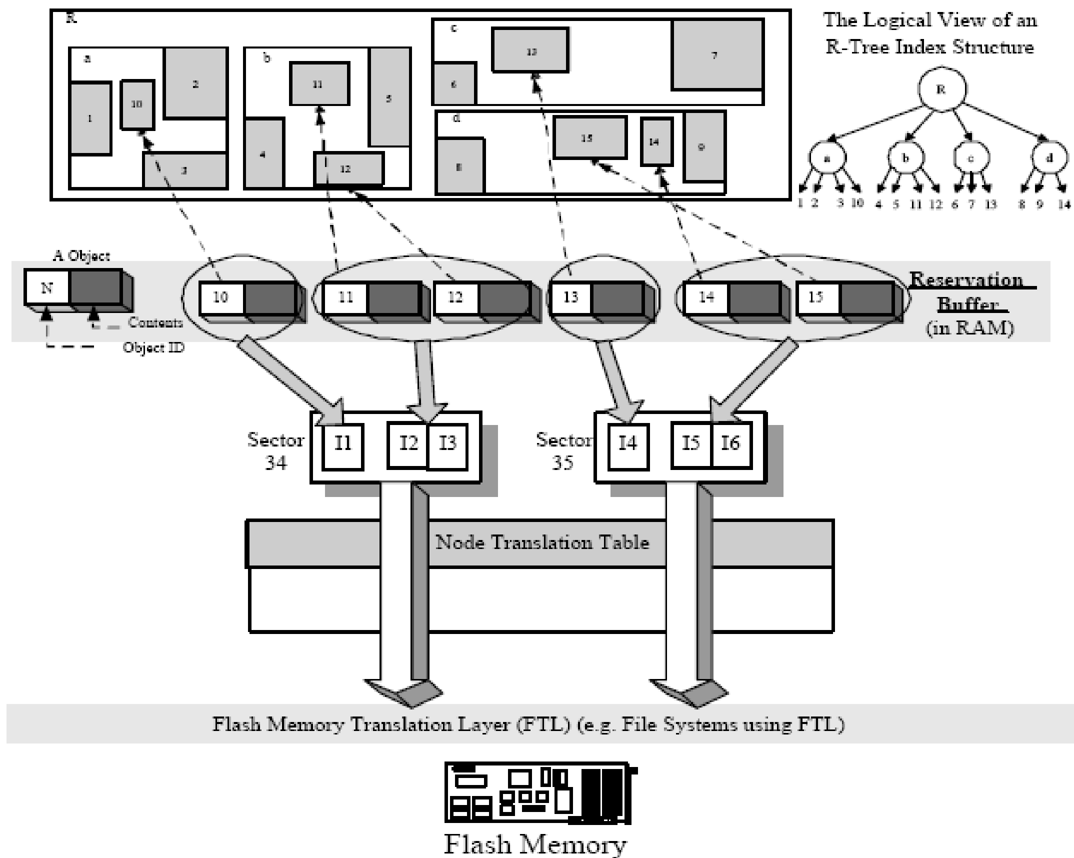
Σχήμα 5.4: Στιγμιότυπο R-δένδρου (μέγιστος βαθμός εξόδου/fanout=5).

Το R-δένδρο του σχήματος 5.4 αποτελείται από έναν εσωτερικό κόμβο *R* και τέσσερις εξωτερικούς κόμβους *a, b, c* και *d.* **Το ελάχιστο επικαλύπτον παραλληλόγραμμο (minimal bounding box)** του κόμβου *R* περιέχει τα ελάχιστα επικαλύπτοντα παραλληλόγραμμα των κόμβων *a, b, c* και *d.* Η ιεραρχική δομή του R-δένδρου παρουσιάζεται στο δεξιό μέρος του σχήματος 5.4.

Η πράξη της εισαγωγής πρέπει να λάβει υπόψιν τα ελάχιστα επικαλύπτοντα παραλληλόγραμμα των κόμβων του R-δένδρου. Το πρώτο αντικείμενο εισάγεται στον κόμβο *a*, το δεύτερο και το τρίτο στον κόμβο *b*, το τέταρτο στον κόμβο *c* και το πέμπτο και το έκτο στον κόμβο *d.* Υποθέτουμε ότι κάθε κόμβος του R-δένδρου αποθηκεύεται σε μία ξεχωριστή σελίδα. Συνεπώς, προκαλούνται έξι ανανεώσεις κόμβων R-δένδρου (δηλαδή, έξι ανανεώσεις σελίδων). Τέτοιου είδους τροποποιήσεις εσωτερικών ή εξωτερικών κόμβων, θεωρούνται αποδοτικές σε αποθηκευτικά συστήματα σκληρών δίσκων, μιας και οι ανανεώσεις κόμβων λαμβάνουν χώρα (localized) στους αντίστοιχους κόμβους. Εξαιτίας όμως των χαρακτηριστικών της μνήμης φλας, οι ανανεώσεις κόμβων πραγματοποιούνται με εκτός-θέσης εγγραφές (out-place writings). Ακόμα και αν τροποποιηθεί ένα μικρό μέρος του κόμβου, η πληροφορία που αποθηκεύεται στον κόμβο πρέπει να γίνει μη έγκυρη (invalidated), και πρέπει να βρεθεί μία νέα σελίδα για τα ανανεωμένα δεδομένα. Οι ανανεώσεις εκτός θέσης, που προκαλούνται από πράξεις πάνω στο R-δένδρο, οδηγούν στη γρήγορη κατανάλωση ελεύθερων σελίδων και σε πολλές περιπτώσεις προκαλούν την εμφάνιση του μηχανισμού συλλογής απορριμμάτων. Ο μηχανισμός συλλογής απορριμμάτων με τη σειρά του, μπορεί να οδηγήσει σε αύξηση της κατανάλωσης ενέργειας μιας και οι λειτουργίες εγγραφής και σβησίματος, καταναλώνουν πολύ περισσότερη ενέργεια από τη λειτουργία ανάγνωσης. Η διατήρηση R-δένδρων στη μνήμη φλας είναι αρκετά περίπλοκη διαδικασία, ειδικά όταν πραγματοποιούνται επαναζυγιστικές πράξεις (όπως διάσπαση ή συγχώνευση κόμβων), οι οποίες προκαλούν την ανανέωση και των δεικτών των κόμβων. Αυτό έχει ως αποτέλεσμα να καταναλώνεται περισσότερος ελεύθερος χώρος και η συλλογή απορριμμάτων να γίνεται συχνότερα.

Όλα τα παραπάνω, οδήγησαν στη δημιουργία μίας αποδοτικής υλοποίησης των R-δένδρων, κατάλληλη για μνήμη φλας. Ο σκοπός της υλοποίησης δεν είναι μόνο η βελτίωση της απόδοσης του συστήματος αλλά και η ελάττωση της κατανάλωσης ενέργειας.

5.3.2 Επισκόπηση της υλοποίησης



Σχήμα 5.5: Αρχιτεκτονική συστήματος.

Η υλοποίηση των R-δένδρων είναι ανεξάρτητη του FTL και των εφαρμογών, όπως φαίνεται και στο σχήμα 5.5. Επίσης, λαμβάνει υπόψιν της τα χαρακτηριστικά της μνήμης φλας, και έχει ως πρωταρχικό σκοπό την παροχή διαφανής και αποδοτικής προσπέλασης πάνω σε R-δένδρα μνήμης φλας, έτσι ώστε να περιορίζονται οι περιττές εγγραφές και να βελτιώνεται η απόδοσητου συσήματος.

Όταν μία εφαρμογή, αιτείται μία πράξη εισαγωγής ή διαγραφής, τότε δημιουργείται ένα 'αντικείμενο' (object) το οποίο περιέχει την αντίστοιχη πράξη καθώς και τα ανάλογα δεδομένα. Το αντικείμενο, αποθηκεύεται προσωρινά σε μία **ειδική περιοχή προσωρινής αποθήκευσης (reservation buffer)**. Τα αντικείμενα που σχετίζονται με πράγεις εισαγωγής και διαγραφής, δημιουργούνται και διαχειρίζονται με τον ίδιο ακριβώς τρόπο. Η ειδική περιοχή προσωρινής αποθήκευσης, είναι μία περιοχή εγγραφής που βρίσκεται στη κύρια μνήμη, όπως φαίνεται και στο σχήμα 5.5. Όλα τα αντικείμενα της ειδικής περιοχής προσωρινής αποθήκευσης, μεταφέρονται στη μνήμη φλας ανά τακτά χρονικά διαστήματα.

Κάθε αντικείμενο της ειδικής περιοχής προσωρινής αποθήκευσης, αποτελείται από δύο μέρη: μεταδεδομένα (meta data) και δεδομένα (data). Τα μεταδεδομένα ενός αντικειμένου, περιλαμβάνουν το ελάχιστο επικαλύπτον παραλληλόγραμμο (minimum bounding box), την αντίστοιχη πράξη (operation) καθώς και δείκτες για τη διατήρηση του αντικειμένου στον reservation buffer. Αξίζει να αναφέρουμε ότι τα περισσότερα

από τα μεταδεδομένα που αναφέρθηκαν παραπάνω, αποθηκεύονται παραδοσιακά σε κόμβους του R-δένδρου, συγκροτώντας έτσι την δομή ευρετηρίου. Μία δομή δεδομένων που ονομάζεται **μονάδα ευρετηρίου (index unit)**, χρησιμοποιείται για την αποθήκευση των μεταδεδομένων ενός αντικειμένου. Όταν μία συλλογή αντικειμένων μεταφέρεται από την ειδική περιοχή προσωρινής αποθήκευσης στη μνήμη φλας, δημιουργούνται οι αντίστοιχες μονάδες ευρετηρίου και τοποθετούνται σε τομείς της μνήμης φλας. Η υπό εξέταση υλοποίηση των R-δένδρων, είναι υπεύθυνη για την τοποθέτηση των μονάδων ευρετηρίου σε ένα μικρό αριθμό τομέων, καθώς και για την αποθήκευση τους στη μνήμη φλας μέσω του FTL. Επειδή οι μονάδες ευρετηρίου ενός τομέα μπορεί να ανήκουν σε διαφορετικούς κόμβους του R-δένδρου, χρησιμοποιείται ένας **πίνακας μετάφρασης κόμβων (node translation table)**, για την διατήρηση των αντίστοιχων θέσεων των μονάδων ευρετηρίου του R-δένδρου.

5.3.3 Δομές δεδομένων

Η συγκεκριμένη υλοποίηση των R-δένδρων, χρησιμοποιεί τρεις δομές δεδομένων. Αξίζει να αναφέρουμε ότι οι δομές που θα παρουσιαστούν, είναι παρόμοιες με αυτές του BFTL, που αναλύθηκε στο κεφάλαιο 4.

- **Ειδική περιοχή προσωρινής αποθήκευσης (Reservation buffer)**: Η ειδική περιοχή προσωρινής αποθήκευσης, είναι μία περιοχή εγγραφής που βρίσκεται στη κύρια μνήμη. Όταν ένας κόμβος R-δένδρου, εισάγεται, διαγράφεται ή τροποποιείται, τα νεοσύστατα αντικείμενα αποθηκεύονται προσωρινά στην ειδική περιοχή προσωρινής αποθήκευσης. Ένα αντικείμενο είναι μία καταχώριση ενός κόμβου R-δένδρου. Τα αντικείμενα της ειδικής περιοχής προσωρινής αποθήκευσης αντιπροσωπεύουν λειτουργίες/πράξεις που δεν έχουν ακόμη εφαρμοστεί στο R-δένδρο.

- **Μονάδες ευρετηρίου (Index units)**: Η φυσική αναπαράσταση ενός κόμβου R-δένδρου, αποτελείται από μονάδες ευρετηρίου. Οι μονάδες ευρετηρίου δημιουργούνται όταν τα αντικείμενα της ειδικής περιοχής προσωρινής αποθήκευσης μεταφέρονται στο R-δένδρο. Μία μονάδα ευρετηρίου περιλαμβάνει τα απαραίτητα μεταδεδομένα ενός αντικειμένου, που αντανακλούν τις τροποποιήσεις στον αντίστοιχο κόμβο του R-δένδρου. Πιο συγκεκριμένα, μία μονάδα ευρετηρίου περιέχει τα παρακάτω συστατικά: data_ptr (δείκτης προς τα δεδομένα), parent_node (δείκτης προς τον κόμβο πατέρα), next_node (δείκτης προς τον κόμβο αδελφό), id (καθορίζει τον κόμβο του R-δένδρου στον οποίο ανήκει η μονάδα ευρετηρίου), minimal_bounding_box (ελάχιστο επικαλύπτον παραλληλόγραμμο) καθώς και op_flag (καθορίζει την ανάλογη πράξη). Αν πρόκειται για εισαγωγή, τότε op_flag=i, αν πρόκειται για διαγραφή, τότε op_flag=d ενώ αν πρόκειται για ανανέωση, τότε op_flag=u. Οι μονάδες ευρετηρίου τοποθετούνται σε λίγους τομείς της μνήμης φλας για την αποφυγή πολλών ανανεώσεων σε επίπεδο R-δένδρου. Έτσι όμως, οι μονάδες ευρετηρίου ενός κόμβου, μπορεί να διασκορπιστούν σε πολλούς διαφορετικούς τομείς της μνήμης φλας.

- **Πίνακας μετάφρασης κόμβων (Node translation table)**: Ο πίνακας μετάφρασης κόμβων είναι ένας πίνακας από λίστες. Κάθε λίστα περιέχει τις

λογικές διευθύνσεις (LBA's) των τομέων που περιέχουν τις μονάδες ευρετηρίου του αντίστοιχου κόμβου του R-δένδρου.

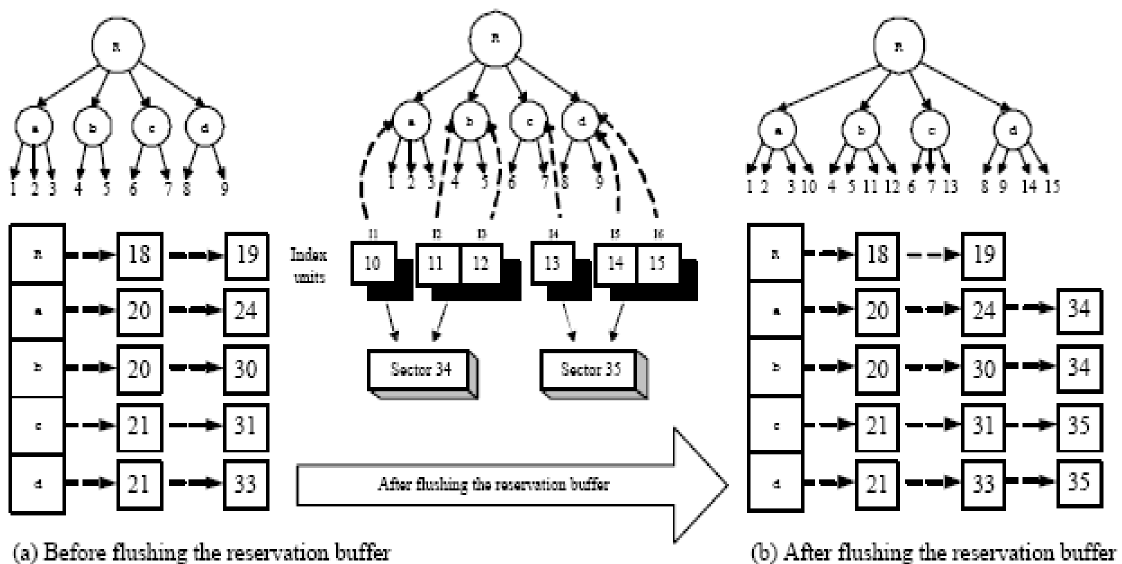
Τοποθέτηση μονάδων ευρετηρίου σε τομείς της μνήμης φλας

Η υπό εξέταση υλοποίηση των R-δένδρων, τοποθετεί τις μονάδες ευρετηρίου σε όσο το δυνατόν λιγότερους τομείς και στη συνέχεια τις αποθηκεύει στη μνήμη φλας μέσω του FTL. Η παραπάνω διαδικασία παρουσιάζεται μέσω ενός παραδείγματος.

Υποθέτουμε ότι η ειδική περιοχή προσωρινής αποθήκευσης μπορεί να αποθηκεύσει μέχρι έξι αντικείμενα με αναγνωριστικά 10, 11, 12, 13, 14 και 15, όπως φαίνεται στο σχήμα 5.5. Από τη στιγμη που η ειδική περιοχή προσωρινής αποθήκευσης έχει γεμίσει, τα έξι αντικείμενα μετατρέπονται σε έξι μονάδες ευρετηρίου {I1, I2, I3, I4, I5, I6}. Σύμφωνα με τα ελάχιστα επικαλύπτοντα παραλληλόγραμμα των αντικειμένων, καθώς και τα ελάχιστα επικαλύπτοντα παραλληλόγραμμα των κόμβων φύλλων *a, b, c, d*, οι έξι μονάδες ευρετηρίου τοποθετούνται σε τέσσερα ξένα σύνολα, για την αποφυγή του διασκορπισμού των μονάδων ευρετηρίου ενός κόμβου σε πολλούς τομείς της μνήμης φλας. Έτσι έχουμε, $\{I1\} \in a$, $\{I2, I3\} \in b$, $\{I4\} \in c$, $\{I5, I6\} \in d$. Υποθέτουμε ότι κάθε τομέας μπορεί να αποθηκεύσει μέχρι τρεις μονάδες ευρετηρίου. {I1} και {I2, I3} αποθηκεύονται στον τομέα 34, ενώ τα {I4} και {I5, I6} αποθηκεύονται στον τομέα 35. Οι δύο τομείς γράφονται στη συνέχεια στη μνήμη φλας. Με την παραδοσιακή προσέγγιση των R-δένδρων, θα χρειαζόμασταν 6 ανανεώσεις κόμβων (δηλαδή έξι εγγραφές τομέων).

Ορισμός 1. Τοποθέτηση μονάδων ευρετηρίου σε τομείς: Δοθείσης μίας συλλογής *T* ξένων συνόλων με μονάδες ευρετηρίου, μίας παραμέτρου χωρητικότητας *C* των τομέων και ενός θετικού ακεραίου *I*, το πρόβλημα της τοποθέτησης μονάδων ευρετηρίου σε τομείς, ανάγεται στη εύρεση ξένων συνόλων ώστε ο συνολικός αριθμός μονάδων ευρετηρίου σε κάθε σύνολο να μην υπερβαίνει το *C*, και το πλήθος των συνόλων να μην υπερβαίνει το *I*.

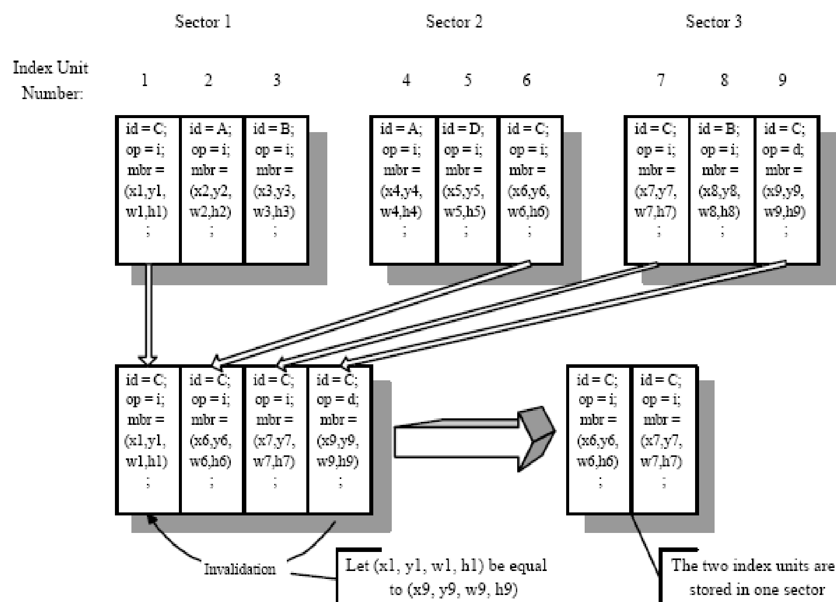
Ανανέωση πίνακα μετάφρασης κόμβων



Σχήμα 5.6: Κατάσταση μετά τη μεταφορά των περιεχομένων του reservation buffer.

Το σχήμα 5.6 (α) παρουσιάζει ένα R-δένδρο με πέντε κόμβους καθώς και τον αντίστοιχο πίνακα μετάφρασης κόμβων. Το σχήμα 5.6 (β) παρουσιάζει το R-δένδρο και τον αντίστοιχο πίνακα μετάφρασης κόμβων, μετά την μεταφορά των περιεχομένων του reservation buffer του σχήματος 5.5. Όταν επισκεπτόμαστε έναν κόμβο του R-δένδρου, συλλέγουμε όλες τις σχετιζόμενες με τον κόμβο μονάδες ευρετηρίου, σαρώνοντας τους τομείς των οποίων οι λογικές διευθύνσεις (LBA's) είναι αποθηκευμένες στη λίστα του πίνακα μετάφρασης κόμβων. Για παράδειγμα, σύμφωνα με το σχήμα 5.6 (β), για την ανακατασκευή του κόμβου *a*, πρέπει να προσπελάσουμε τις μονάδες ευρετηρίου στους τομείς με λογικές διευθύνσεις 20, 24 και 34. Από την άλλη πλευρά, ένας τομέας, για παράδειγμα αυτός με λογική διεύθυνση 20, μπορεί να περιέχει μονάδες ευρετηρίου για περισσότερους από έναν κόμβο (για τον *a* και *b*).

Παρ'όλα αυτά, οι λίστες του πίνακα μετάφρασης κόμβων μπορούν να μεγαλώσουν απροσδόκητα. Για παράδειγμα, εάν η λίστα μίας εγγραφής του πίνακα μετάφρασης κόμβων, αποτελείται από 100 στοιχεία (αποθηκεύει 100 διευθύνσεις τομέων), τότε η επίσκεψη του αντίστοιχου κόμβου απαιτεί 100 αναγνώσεις τομέων. Κάτι τέτοιο όμως, μπορεί να μειώσει σε μεγάλο βαθμό την απόδοση του συστήματος καθώς και να καταναλώσει μεγάλες ποσότητες μνήμης RAM. Για την αντιμετώπιση του παραπάνω προβλήματος, προτείνεται η **συμπίεση (compaction)** του πίνακα μετάφρασης κόμβων, όποτε είναι απαραίτητο. Μία παράμετρος συστήματος *w*, χρησιμοποιείται για να ελέγχει το μέγιστο μέγεθος των λιστών του πίνακα μετάφρασης κόμβων. Πιο συγκεκριμένα, όταν το μέγεθος μίας λίστας υπερβαίνει την παράμετρο *w*, τότε η λίστα συμπιέζεται. Για τη συμπίεση μίας λίστας, διαβάζονται όλοι οι τομείς που περιέχουν τις μονάδες ευρετηρίου του αντίστοιχου κόμβου, και στη συνέχεια οι μονάδες ευρετηρίου γράφονται στη μνήμη φλας, σε όσο το δυνατόν μικρότερο αριθμό τομέων. Από τη στιγμή που η πράξη της διαγραφής αντιμετωπίζεται ως ένα είδος πράξης εισαγωγής στην συγκεκριμένη υλοποίηση, χρησιμοποιώντας μη έγκυρα αντικείμενα (invalidation objects) στην ειδική περιοχή προσωρινής αποθήκευσης, η συμπίεση του πίνακα μετάφρασης κόμβων, ελαττώνει τις περιττές μονάδες ευρετηρίου. Το παράδειγμα που ακολουθεί, παρουσιάζει τη διαδικασία της διάσπασης:



Σχήμα 5.7: Συμπίεση κόμβου *C* του R-δένδρου.

Στο σχήμα 5.7 μερικές μονάδες ευρετηρίου του κόμβου *C* διασκορπίζονται στους τομείς 1, 2 και 3. Έστω ότι κάθε τομέας χωράει τρεις μονάδες ευρετηρίου. Έτσι, αποθηκεύονται εννέα μονάδες ευρετηρίου στους τρεις τομείς, αριθμημένοι από 1 έως 9. Έστω ότι με (x_i, y_i) συμβολίζουμε τις συντεταγμένες της χαμηλώτερης αριστερής γωνίας, και με (w_i, h_i) τις διαστάσεις (πλάτος, ύψος) του ελάχιστου επικαλύπτοντος παραλληλογράμμου αντίστοιχα, κάθε μονάδας ευρετηρίου (i=1,…,9). Η συμπίεση της λίστας που σχετίζεται με τον κόμβο *C*, περιλαμβάνει την ανάγνωση των τριών τομέων και την εγγραφή των μονάδων ευρετηρίου. Κατά τη διάρκεια της συμπίεσης, το σύστημα ανακαλύπτει ότι η πρώτη (εισαγωγή) και η έννατη (διαγραφή) μονάδα ευρετηρίου, έχουν το ίδιο ελάχιστο επικαλύπτον παραλληλόγραμμο. Αυτό έχει ως αποτέλεσμα, η πράξη της διαγραφής να αφαιρεί (removing) την πράξη της εισαγωγής. Έτσι, η έκτη και η έβδομη μονάδα ευρετηρίου του κόμβου *C*, αποθηκεύονται σε έναν κοινό τομέα. Ο πίνακας μετάφρασης κόμβων τροποποιείται αναλόγως.

5.3.4 Ανάλυση πολυπλοκότητας

Υποθέτουμε ότι πρέπει να εισάγουμε *n* γεωμετρικά δεδομένα με διαφορετικά ελάχιστα επικαλύπτοντα παραλληλόγραμμα. Έστω ένα R-δένδρο μνήμης φλας, με ύψος *H*, στο οποίο κάθε κόμβος αποθηκεύεται σε έναν ξεχωριστό τομέα της μνήμης φλας. Το ύψος *H*, φράσεται από το όριο $O(\log_{fan-out}(m+n))$, όπου με *m* συμβολίζουμε τον αριθμό των αντικειμένων προτού εισάγουμε τα *n* αντικείμενα. Αρχικά, θα υπολογίσουμε το πλήθος των τομέων που προσπελαύνονται κατά την εισαγωγή των *n* γεωμετρικών δεδομένων, σε ένα παραδοσιακό R-δένδρο.

Υποθέτουμε ότι το πλήθος της διάσπασης κόμβων ισούται με N_{split} κατά την εισαγωγή *n* γεωμετρικών δεδομένών. Ο αριθμός των αναγνώσεων και των εγγραφών κατά την εισαγωγή είναι $R_R = O(n*H)$ και $W_R = O(n+3*N_{split})$, αντίστοιχα, σε ένα παραδοσιακό R-δένδρο.

Ο αριθμός των τομέων που διαβάζονται κατά την εισαγωγή *n* γεωμετρικών δεδομένων, στην υλοποίηση των R-δένδρων για μνήμη φλας, είναι ίσος με: $R_{PR} = O(n*H*w)$, μιάς και η επίσκεψη ενός κόμβου απαιτεί την διαπέραση μίας λίστας διευθύνσεων στον πίνακα μετάφρασης κόμβων. Αν συγκρίνουμε τον αριθμό R_{PR} με τον αριθμό R_R, οδηγούμαστε στο συμπέρασμα ότι η υλοποίηση των R-δένδρων για μνήμη φλας, διαβάζει περισσότερους τομείς απ'ότι τα παραδοσιακά R-δένδρα, κατά την εισαγωγή *n* αντικειμένων. Στην πραγματικότητα όμως, υλοποίηση των R-δένδρων για μνήμη φλας, ανταλλάσσει τον αριθμό των εγγραφών με τον αριθμό των αναγνώσεων.

Ο αριθμός των τομέων που γράφονται κατά την εισαγωγή *n* γεωμετρικών δεδομένων, στην υλοποίηση των R-δένδρων για μνήμη φλας, είναι ίσος με: $W_{PR} = O(2*(\sum_{i=1}^{\lceil n/b \rceil}\frac{b}{\Lambda}) + N_{split}) = O(\frac{2*n}{\Lambda} + N_{split})$, όπου με *b* συμβολίζουμε την χωρητικότητα του reservation buffer και Λ=(fanout-1). Παρατηρούμε ότι ο αριθμός

W_{PR} είναι πολύ μικρότερος του W_R, από τη στιγμή που το Λ (μέγιστος αριθμός μονάδων ευρετηρίου σε έναν τομέα), είναι συνήθως μεγαλύτερο του 2.

Βιβλιογραφία κεφαλαίου

[1] A. Guttman. R-tree: A dynamic index structure for spatial searching, 1984. (χρήση στις σελίδες 90-98)

[2] Chin-Hsien Wu, Li-Oin Chang, Tei-Wei Kuo. An efficient R-tree implementation over flash memory storage systems 2003. (χρήση στις σελίδες 98-104)

Κεφάλαιο 6

Μηχανισμοί εντοπισμού «καυτών» (hot) δεδομένων

Περιεχόμενα

6.1 Εισαγωγή………………………………………………………........... 107

6.2 Μηχανισμός πολλαπλών συναρτήσεων κατακερματισμού............... 107

6.2.1 Βελτίωση της απόδοσης του μηχανισμού... 110

Βιβλιογραφία κεφαλαίου... 110

6.1 Εισαγωγή

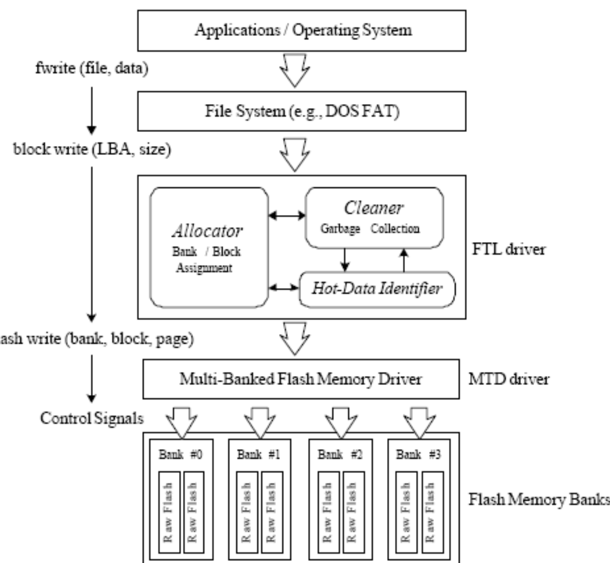
Τα τελευταία χρόνια, η μνήμη φλας έχει καθιερωθεί ως μία εξαιρετική εναλλακτική λύση, κατά τον σχεδιασμό και την υλοποίηση του αποθηκευτικού συστήματος των ενσωματωμένων συστημάτων. Εξαιτίας της πολύ περιορισμένης υπολογιστικής ισχύς των μικροεπεξεργαστών των ενσωματωμένων συστημάτων, κρίνεται απαραίτητος ο σχεδιασμός αποδοτικών μεθόδων για την διαχείριση του αποθηκευτικού χώρου. Προς αυτή την κατεύθυνση κινούνται και οι **μηχανισμοί εντοπισμού «καυτών» (hot) δεδομένων**.

Σε έναν μηχανισμό εντοπισμού «καυτών» δεδομένων, μία λογική διεύθυνση μπλοκ (LBA) εξετάζεται για να διαπιστωθεί το κατά πόσο περιέχει **συχνά προσπελάσιμα/συχνά ανανεώσιμα (frequently accessed/frequently updated) δεδομένα (εξ ου και ο όρος καυτά δεδομένα)**. Ο εντοπισμός των «καυτών» δεδομένων, επιδρά στη συχνότητα εμφάνισης του μηχανισμού συλλογής απορριμμάτων, ενώ παράλληλα αυξάνει τη διάρκεια ζωής της μνήμης φλας.

Στις ενότητες που ακολουθούν, θα παρουσιαστεί μία αποδοτική μέθοδος εντοπισμού «καυτών» δεδομένων, με μικρές απαιτήσεις μνήμης. Πιο συγκεκριμένα, αναλύεται ένας **μηχανισμός πολλαπλών συναρτήσεων κατακερματισμού**, στον οποίο πολλαπλές ανεξάρτητες συναρτήσεις κατακερματισμού μειώνουν την πιθανότητα εσφαλμένου εντοπισμού «καυτών» δεδομένων, ενώ παράλληλα παρέχουν εξαιρετική απόδοση σωστής αναγνώρισης «καυτών» δεδομένων.

6.2 Μηχανισμός πολλαπλών συναρτήσεων κατακερματισμού

Στην ενότητα αυτή θα παρουσιάσουμε έναν μηχανισμό που βασίζεται σε συναρτήσεις κατακερματισμού (hash functions), για τον εντοπισμό «καυτών» δεδομένων. Ο μηχανισμός αυτός είναι γνωστός ως ***αναγνωριστής «καυτών» δεδομένων (hot-data identifier)***. Η υλοποίηση του μηχανισμού πραγματοποιείται στο στρώμα του FTL, όπως φαίνεται και στο σχήμα 6.1 που ακολουθεί.



Σχήμα 6.1: Ενσωμάτωση *αναγνωριστή «καυτών» δεδομένων* στο FTL.

Ο πίνακας 6.1 που ακολουθεί, περιλαμβάνει τους ορισμούς των παραμέτρων που χρησιμοποιούνται στον υπό εξέταση μηχανισμό.

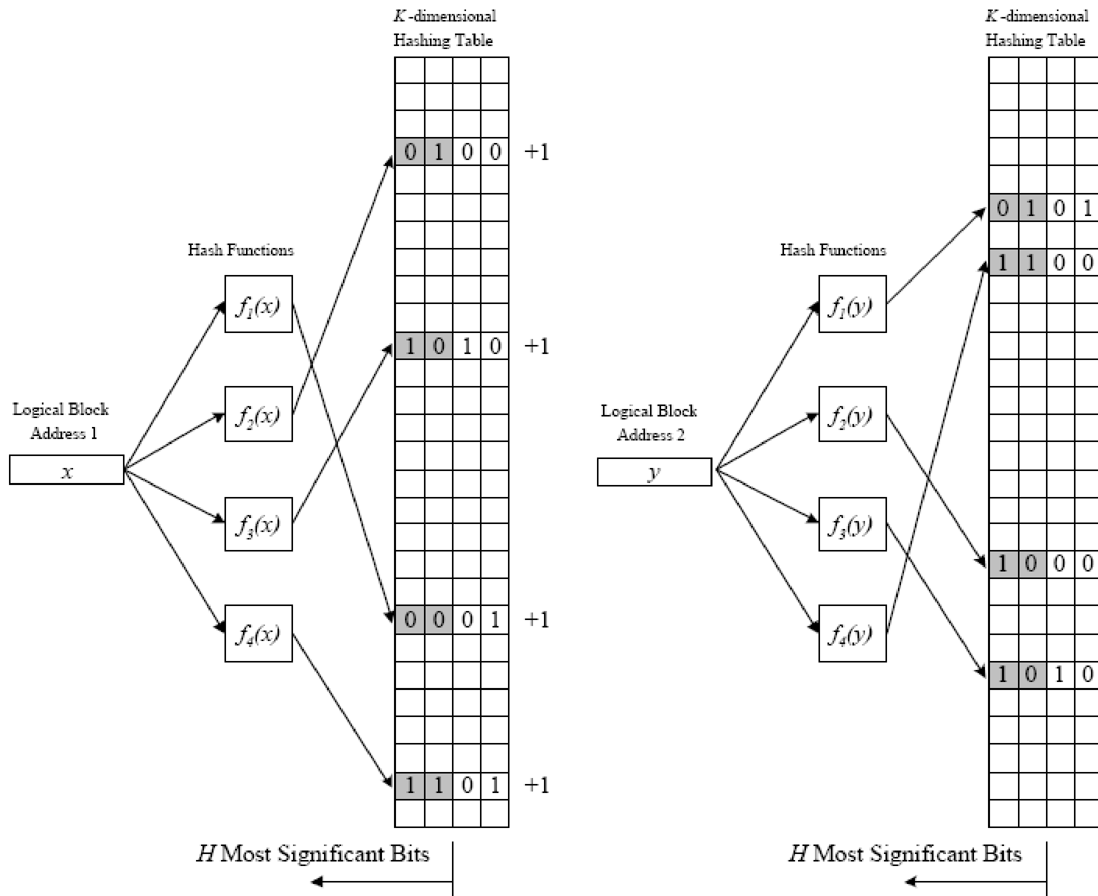
System Model Parameters	Notation
Number of Hash Functions	K
Size of Counter	C
Write Counts for an LBA to Become Hot	$2^{(C-H)}$
Number of Counters in a Hash Table	M
Number of Write References	N
Ratio of Hot Data in All Data ($< 50\%$)	R

Πίνακας 6.1: Παράμετροι συστήματος.

Ο μηχανισμός χρησιμοποιεί *K* ανεξάρτητες συναρτήσεις κατακερματισμού, για να κατακερματίσει (hash) μία δοθείσα λογική διεύθυνση μπλοκ (LBA) της μνήμης φλας, σε πολλαπλές καταχωρίσεις (entries) ενός πίνακα κατακερματισμού (hash table) *M* θέσεων. Κάθε καταχώριση του πίνακα κατακερματισμού, συσχετίζεται με έναν μετρητή των *C* bits, για να καταμετράται το πλήθος των εγγραφών που λαμβάνουν χώρα στη συγκεκριμένη λογική διεύθυνση. Πιο συγκεκριμένα, όταν μία πράξη εγγραφής προωθείται προς το FTL, η αντίστοιχη λογική διεύθυνση μπλοκ, κατακερματίζεται ταυτόχρονα από τις *K* ανεξάρτητες συναρτήσεις κατακερματισμού. Οι μετρητές των *K* καταχωρίσεων του πίνακα κατακερματισμού που σχετίζονται με τη συγκεκριμένη LBA, αυξάνονται κατά μία μονάδα, αντανακλώντας με αυτόν τον τρόπο το γεγονός ότι η πραγματοποιήθηκε εγγραφή στη συγκεκριμένη LBA. Εάν ένας μετρητής φτάσει τη μέγιστη τιμή του, τότε παραμένει αμετάβλητος. Στο σημείο αυτό πρέπει να αναφέρουμε ότι δεν γίνεται καμία αύξηση μετρητών, όταν πρόκειται για λειτουργίες ανάγνωσης, αφού δε χρειάζεται να ανακηρυχτεί καμία σελίδα ως μη έγκυρη.

Για κάθε δοθέντα αριθμό τομέων που έχουν γραφτεί, αποκαλούμε ως **περίοδο φθοράς (decay period)** των αριθμών εγγραφής, τη διαδικασία κατά την οποία οι τιμές των μετρητών διαιρούνται διά 2, έτσι ώστε να μειώσουμε εκθετικά τις τιμές των αριθμών εγγραφής, καθώς περνάει ο χρόνος. Όταν μία λογική διεύθυνση μπλοκ, πρέπει να εξεταστεί για το κατά πόσο περιέχει «καυτά» δεδομένα, τότε η διεύθυνση κατακερματίζεται ταυτόχρονα από τις *K* ανεξάρτητες συναρτήσεις κατακερματισμού. Η λογική διεύθυνση μπλοκ, περιέχει «καυτά» δεδομένα εάν τα *H* σημαντικότερα bits κάθε μετρητή των *K* κατακερματισμένων τιμών, περιέχουν ένα μη μηδενικό bit.

Το σχήμα 6.2 (α), παρουσιάζει την αύξηση των μετρητών των *K* καταχωρίσεων του πίνακα κατακερματισμού μίας δοθείσης LBA, όπου υπάρχουν τέσσερις ανεξάρτητες συναρτήσεις κατακρματισμού (*K=4*), και κάθε μετρητής αποτελείται από τέσσερα bits (*C=4*).



(α) Αύξηση μετρητών μίας LBA. (β) Εντοπισμός «καυτών» δεδομένων μίας LBA.

Σχήμα 6.2: (α), (β). Αύξηση μετρητών και εντοπισμός «καυτών» δεδομένων μίας LBA, όπου *C=4, K=4 και H=2*.

Το σχήμα 6.2 (β) παρουσιάζει τον εντοπισμό «καυτών» δεδομένων μίας LBA. Μόνο τα δύο σημαντικότερα bits κάθε μετρητή εξετάζονται, για να διαπιστωθεί το κατά πόσο η λογική διεύθυνση μπλοκ, περιέχει «καυτά» δεδομένα. Στο συγκεκριμένο παράδειγμα, η λογική διεύθυνση μπλοκ *y*, περιέχει «καυτά» δεδομένα, μιάς και οι τέσσερις καταχωρίσεις του πίνακα κατακερματισμού, που σχετίζονται με την συγκεκριμένη LBA, περιέχουν τουλάχιστον έναν άσσο, στα δύο σημαντικότερα bits τους, αντίστοιχα.

Ο λόγος που χρησιμοποιούνται *K* ανεξάρτητες συναρτήσεις κατακερματισμού, είναι να μειωθεί η πιθανότητα εσφαλμένου εντοπισμού «καυτών» δεδομένων μίας LBA. Επειδή η τεχνική του κατακερματσμού, τείνει να αντιστοιχίζει τυχαία μία μεγάλη περιοχή διευθύνσεων σε μία μικρότερη, είναι πιθανόν να ανακηρύξουμε λανθασμένα μία LBA ως περιοχή «καυτών» δεδομένων. Με τη χρήση όμως πολλαπλών συναρτήσεων κατακερματισμού, η πιθανότητα εσφαλμένου εντοπισμού «καυτών» δεδομένων μειώνεται αισθητά.

6.2.1 Βελτίωση της απόδοσης του μηχανισμού

Στην ενότητα αυτή θα παρουσιατεί μία μέθοδος βελτίωσης της απόδοσης του υπό εξέταση μηχανισμού, όσον αφορά τον εσφαλμένο εντοπισμό «καυτών» δεδομένων. Πιο συγκεκριμένα, αντί να μεγαλώνουμε τον πίνακα κατακερματισμού για την βελτίωση του εσφαλμένου εντοπισμού «καυτών» δεδομένων, προτείνεται η αύξηση των μετρητών των K κατακερματισμένων τιμών μίας LBA, που έχουν την μικρότερη τιμή. Ο λόγος της παραπάνω πολιτικής προαύξησης των μετρητών, είναι ο εξής: η αιτία του εσφαλμένου εντοπισμού «καυτών» δεδομένων είναι το ότι οι K μετρητές μίας μη «καυτής» LBA, αυξάνονται κάτω από εγγραφές μη «καυτών» δεδομένων, εξαιτίας συκγρούσεων κατακερματισμού (hashing collision). Από την άλλη πλευρά, εάν μία LBA περιέχει «καυτά» δεδομένα, η παραπάνω πολιτική προαύξησης των μετρητών θα οδηγούσε τους K μετρητές της LBA να υπερβούν την τιμή $2^{(C-H)}$ (επειδή κάποιες άλλες εγγραφές θα αναπλήρωναν την απώλεια προαύξησης των μετρητών). Παρ'όλα αυτά, εάν μία LBA δεν περιέχει «καυτά» δεδομένα, τότε η προτεινόμενη πολιτική προαύξησης μετρητών, θα μείωνε την πιθανότητα εσφαλμένου εντοπισμού, μιας και θα αυξανόταν εσφαλμένα, ένας μικρότερος αριθμός μετρητών εξαιτίας της σύγκρουσης κατακερματισμού.

Βιβλιογραφία κεφαλαίου

[1] Jen-Wei Hsieh, Li-Pin Chang, Tei-Wei Kuo. Efficient On-line Identification of Hot Data for Flash-Memory Management. (χρήση στις σελίδες 107-110)

Κεφάλαιο 7

Μηχανισμοί συλλογής απορριμμάτων και εξισορρόπησης φθοράς

Περιεχόμενα

7.1 Εισαγωγή…………………………………………………………......... 112

7.2 Εξισορρόπηση φθοράς βασισμένη στην ομαδοποίηση

(Group-based wear leveling)………………………………………. 113

7.2.1 Αλγόριθμος ανταλλαγής «καυτών» - «παγωμένων» δεδομένων

(Hot-cold swapping algorithm)……………………………………...113

7.2.2 Αλγόριθμος βασισμένος στην ομαδοποίηση

(Group-based algorithm)……………………………………………..114

7.2.3 Αποτρέποντας την εσφαλμένη ανταλλαγή..116

7.2.4 Καθορίζοντας τις τιμές των *TH* και *λ*...117

7.2.5 Ανάλυση χωρικής πολυπλοκότητας.. 118

7.3 Αποδοτικός μηχανισμός στατικής εξισορρόπησης φθοράς.............118

7.3.1 Επισκόπηση μηχανισμού..……………....118

7.3.2 Πίνακας σβησμένων μπλοκ (BET)……………………………….....119

7.3.3 SW Leveler……………………………………………………….....120

7.3.4 Ανάλυση χωρικής πολυπλοκότητας..122

7.4 Μηχανισμός συλλογής απορριμμάτων πραγματικού χρόνου

(Real-time garbage collection mechanism)……………………....122

7.4.1 Μοντέλο εργασιών πραγματικού χρόνου...123

7.4.2 Συλλέκτες απορριμμάτων πραγματικού χρόνου.................................... 125

7.4.3 Μηχανισμός ανανέωσης ελεύθερων σελίδων...126

7.4.4 Πολιτική ανακύκλωσης μπλοκ... 128

7.4.5 Υποστήριξη εργασιών μη πραγματικού χρόνου......................................129

7.4.5.1 Ανανέωση ελεύθερων σελίδων για εργασίες μη πραγματικού χρόνου...129

7.4.5.2 Εξισορροπιστής φθοράς μη πραγματικού χρόνου..................................130

Βιβλιογραφία κεφαλαίου..130

7.1 Εισαγωγή

Όπως είναι γνωστό από το κεφάλαιο 1, η μνήμη φλας είναι **μονής εγγραφής (write-once)**, με αποτέλεσμα τα νέα δεδομένα να μην πανωγράφουν (overwrite) τα παλαιά σε κάθε διαδικασία ανανέωσης (update). Αντιθέτως, τα νέα δεδομένα (live data) γράφονται στον ελεύθερο/διαθέσιμο χώρο (free space) και οι παλαιές εκδόσεις των δεδομένων (dead data) κηρύσσονται ως μη έγκυρες (invalidated). Η παραπάνω στρατηγική ανανέωσης ονομάζεται **ανανέωση εκτός θέσης (outplace update)**. Με άλλα λόγια, μία σελίδα NAND μνήμης φλας, πρέπει πρώτα να σβηστεί προτού τα νέα δεδομένα (updated data) γραφτούν σε αυτή (και στην ίδια ακριβώς θέση).

Μετά από ένα συγκεκριμένο αριθμό εγγραφών σε επίπεδο σελίδας, ο ελεύθερος χώρος της μνήμης φλας μειώνεται αισθητά. Αυτό έχει ως αποτέλεσμα, την εκκίνηση δραστηριοτήτων, που αποτελούνται από αναγνώσεις (reads), εγγραφές (writes) καθώς και σβησίματα (erases), με σκοπό την **ανάκτηση (reclaim) ελεύθερου αποθηκευτικού χώρου**. Οι δραστηριότητες αυτές συνθέτουν τον **μηχανισμό συλλογής απορριμμάτων (garbage collection mechanism)**, και επιφέρουν επιπλέον φόρτο στη διαχείριση της μνήμης φλας. Ο σκοπός του μηχανισμού συλλογής απορριμμάτων, είναι να ανακυκλώσει (recycle) σελίδες που περιέχουν απαρχαιωμένα δεδομένα (dead pages), έτσι ώστε να μετατραπούν σε ελεύθερες σελίδες (free pages). Μία ***πολιτική ανακύκλωσης μπλοκ (block-recycling policy)***, είναι υπεύθυνη για την επιλογή των κατάλληλων μπλοκ προς σβήσιμο. Η πολιτική ανακύκλωσης μπλοκ, προσπαθεί να ελαχιστοποιήσει το φόρτο που προκαλείται κατά την εμφάνιση του μηχανισμού συλλογής απορριμμάτων.

Για τη δημιουργία ελεύθερου αποθηκευτικού χώρου που θα προορίζεται για τα ανανεωμένα μπλοκ δεδομένων (updated blocks), θα πρέπει να ανακτηθούν οι απαρχαιωμένοι τομείς (obsolete sectors). Από τη στιγμή που ο μόνος τρόπος για την ανάκτηση ενός τομέα είναι να σβήσουμε μία ολόκληρη μονάδα σβησίματος (erase unit), ο μηχανισμός συλλογής απορριμμάτων διαχειρίζεται μονάδες σβησίματος. Η διαδικασία ανάκτησης ελέυθερου αποθηκευτικού χώρου, μπορεί να λάβει χώρα είτε κατά το παρασκήνιο, όπου η φλας συσκευή είναι αδρανής (idle), είτε όταν το ποσοστό ελεύθερου διαθέσιμου αποθηκευτικού χώρου της μνήμης φλας πέφτει κάτω από ένα συγκεκριμένο κατώφλι. Η ανάκτηση ελεύθερου χώρου πραγματοποιείται στα παρακάτω βήματα:

- Μία ή παραπάνω μονάδες σβησίματος επιλέγονται για ανάκτηση.
- Οι έγκυροι τομείς αυτών των μονάδων σβησίματος αντιγράφονται σε κάποια ελέυθερη περιοχή της μνήμης φλας.
- Εάν είναι απαραίτητο, ανανεώνονται οι δομές δεδομένων που αντιστοιχίζουν λογικά μπλοκ σε τομείς, γεγονός που αντανακλά την παραπάνω επανατοποθέτηση των τομέων.
- Τέλος, οι παραπάνω μονάδες σβησίματος, σβήνονται και οι τομείς τους είναι διαθέσιμοι για εκ νέου χρήση.

Ένα μπλοκ μνήμης φλας, έχει έναν περιορισμό στο πλήθος σβησιμάτων που μπορεί να υποστεί. Για παράδειγμα, κάθε μπλοκ μίας τυπικής NAND μνήμης φλας,

μπορεί να σβηστεί (erased) 10^6 φορές. Όταν ξεπεραστεί αυτό το όριο, το μπλοκ φθείρεται (worns out) και υποφέρει από λάθη κατά την εγγραφή (write errors). Ο **μηχανισμός εξισορρόπησης φθοράς (wear leveling mechanism)** προσπαθεί να σβήνει τα μπλοκ της μνήμης φλας με **ομοιόμορφο τρόπο (evenly)**, έτσι ώστε να αυξάνεται η διάρκεια ζωής (life time) της μνήμης φλας. Υπάρχουν δύο βασικές τεχνικές εξισσορόπησης φθοράς. Η **δυναμική εξισσορόπηση φθοράς (dynamic wear leveling)** και η **στατική εξισσορόπηση φθοράς (static wear leveling)**.

Στην δυναμική εξισορρόπηση φθοράς, ανακυκλώνονται μπλοκ με μικρό μετρητή σβησίματος (small erase counter). Σπουδαίο ρόλο στην απόδοση του μηχανισμού δυναμικής εξισορρόπησης φθοράς, κατέχει ο μηχανισμός εντοπισμού «καυτών» δεδομένων, που μελετήσαμε στο κεφάλαιο 6. Ενώ με αυτή την τεχνική εξισορροπείται η φθορά, η βελτίωση της ανθεκτικότητας (endurance improvement) της μνήμης φλας εξαρτάται αποκλειστικά από τη φύση της. Πιο συγκεκριμένα, μπλοκ που περιέχουν **«παγωμένα» (cold) δεδομένα (δεδομένα που ανανεώνονται πολύ σπάνια)**, είναι πιθανό να παραμείνουν ανέπαφα, ανεξάρτητα από το πως οι ανανεώσεις «καυτών» (hot) δεδομένων φθείρουν άλλα μπλοκ δεδομένων. Με άλλα λόγια, οι ανανεώσεις και οι ανακυκλώσεις μπλοκ/σελίδων, πραγματοποιούνται είτε σε ελέυθερα μπλοκ, είτε σε μπλοκ που περιέχουν «καυτά» δεδομένα.

Η στατική εξισορρόπηση φθοράς είναι εκ διαμέτρου αντίθετη από την δυναμική εξισορρόπηση φθοράς. Σκοπός της είναι να προφυλάξει τα «παγωμένα» δεδομένα από το να παραμένουν σε κάποια μπλοκ για μεγάλα χρονικά διαστήματα, έτσι ώστε η εξισορρόπηση φθοράς να εφαρμόζεται ομοιόμορφα σε όλα τα μπλοκ της μνήμης φλας.

7.2 Εξισορρόπηση φθοράς βασισμένη στην ομαδοποίηση (Group-based wear leveling)

7.2.1 Αλγόριθμος ανταλλαγής «καυτών» - «παγωμένων» δεδομένων (Hot-cold swapping algorithm)

Όπως είναι γνωστό από το κεφάλαιο 2, η τεχνική αντιστοίχησης μπλοκ του FTL, μεταφράζει τις λογικές διευθύνσεις των μπλοκ (LBAs) σε φυσικές διευθύνσεις της μνήμης φλας. Κάθε λογική διεύθυνση μπλοκ αποτελείται από έναν λογικό αριθμό μπλοκ (logical block number) και ένα αντιστάθμισμα (offset). Ο λογικός αριθμός του μπλοκ, αντιστοιχίζεται σε ένα φυσικό μπλοκ της μνήμης φλας (το μπλοκ δεδομένων) με τη βοήθεια του πίνακα μετάφρασης διευθύνσεων (address translation table), ενώ το αντιστάθμισμα χρησιμοποιείται για την εύρεση της σελίδας που περιέχει τα επιθυμητά δεδομένα, μέσα στο μπλοκ. Για την ανανέωση δεδομένων, εκχωρείται/ανατίθεται ένα μπλοκ ανανέωσης (update block) (συχνά αποκαλείται μπλοκ αντικατάστασης) στο ανάλογο μπλοκ δεδομένων, και τα νεοεισερχόμενα δεδομένα γράφονται πλέον στο μπλοκ ανανέωσης. Όταν δεν υπάρχουν αρκετά ελεύθερα μπλοκ ανανέωσης προς ανάθεση, τότε ενεργοποιείται η διαδικασία συγχώνευσης για την ανάκτηση ελεύθερων μπλοκ, η οποία συνενώνει ένα μπλοκ δεδομένων με τα σχετιζόμενα μπλοκ ανανέωσης.

Ο **αλγόριθμος εξισορρόπησης φθοράς που βασίζεται στην ομαδοποίηση (group-based wear leveling algorithm)**, στηρίζεται στην ***ανταλλαγή «καυτών» - «παγωμένων» δεδομένων (hot-cold swapping)***. Ο αλγόριθμος ανταλλαγής «καυτών» - «παγωμένων» δεδομένων, έχει ως σκοπό να εξισορροπήσει το πλήθος σβησίματος (erase counter) των μπλοκ, ανταλλάσσοντας περιοδικά τα «καυτά» δεδομένα που βρίσκονται σε παλαιά (old) μπλοκ με «παγωμένα» δεδομένα που βρίσκονται σε νέα (young) μπλοκ.

Στον υπό εξέταση αλγόριθμο, η συνθήκη ανταλλαγής εξετάζεται οποτεδήποτε ένα νέο μπλοκ ανανέωσης ανατίθεται σε ένα μπλοκ δεδομένων. Ένα μπλοκ ανανέωσης θα σβηστεί αρκετά νωρίς, με αποτέλεσμα ο μετρητής σβησίματός (erase counter) του να αυξάνεται γρηγορότερα, σε σύγκριση με ένα μπλοκ που περιέχει «παγωμένα» δεδομένα. Προτού γίνει η ανάθεση του μπλοκ ανανέωσης, γίνεται σύγκριση του μετρητή σβησίματος του μπλοκ ανανέωσης με τον μετρητή σβησίματος του πιο νέου μπλοκ (youngest block). Η αναταλλαγή πραγματοποιείται μόνο όταν η διαφορά μεταξύ των δύο μετρητών σβησίματος είναι μεγαλύτερη από ένα συγκεκριμένο κατώφλι, *TH*. Αντί να γίνεται ανταλλαγή «καυτών» δεδομένων που βρίσκονται σε παλαιά μπλοκ με «παγωμένα» δεδομένα που βρίσκονται σε νέα μπλοκ, ο αλγόριθμος περιορίζεται στο να **μετακινεί «παγωμένα» δεδομένα σε ένα παλαιό μπλοκ**, μειώνοντας με αυτόν τον τρόπο τον φόρτο που προκαλείται στη μνήμη φλας εξαιτίας της ανταλλαγής.

7.2.2 Αλγόριθμος βασισμένος στην ομαδοποίηση (Group-based algorithm)

Ο κύριος σκοπός του αλγορίθμου που βασίζεται στην ομαδοποίηση των μπλοκ, είναι η ελάττωση των απαιτήσεων μνήμης που χρειάζονται, για την αποθήκευση της πληροφορίας σχετικά με την φθορά των μπλοκ της μνήμης φλας (wear information). Η κύρια ιδέα είναι να ομαδοποιήσουμε πολλά μπλοκ σε μία ομάδα (group), και να διατηρούμε στη μνήμη μόνο μία σύνοψη/περίληψη της πληροφορίας φθοράς της ομάδας, αντί να διατηρούμε την πληροφορία φθοράς κάθε ανεξάρτητου μπλοκ.

Στον συγκεκριμένο αλγόριθμο, μία ομάδα αποτελείται από έναν συγκεκριμένο αριθμό συνεχόμενων λογικών μπλοκ, ενώ η περίληψη της ομάδας (group summary) περιέχει πληροφορία σχετικά με την ολική κατάσταση φθοράς της ομάδας. Αυτή η πληροφορία, χρησιμοποιείται για τον εντοπισμό άνισης φθοράς μεταξύ των μπλοκ, καθώς και για την εύρεση ομάδων «θυμάτων» (victim groups) στις οποίες θα εφαρμοστεί ο μηχανισμός εξισορρόπησης φθοράς. Ας υποθέσουμε ότι η περίληψη της ομάδας, δείχνει ότι τα μπλοκ της συγκεκριμένης ομάδας είναι αρκετά νεότερα σε σχέση με άλλα μπλοκ. Τότε, η συγκεκριμένη ομάδα επιλέγεται ως «θύμα» για εξισορρόπηση φθοράς, και ένα μπλοκ της ομάδας ανταλλάσσεται με ένα παλαιό μπλοκ.

Από τη στιγμή που ο αλγόριθμος βασίζεται στην πληροφορία που βρίσκεται στην περίληψη της ομάδας, τότε η αποδοτικότητα του μηχανισμού εξισορρόπησης φθοράς εξαρτάται σε πολύ μεγάλο βαθμό από την ακρίβεια της παραπάνω πληροφορίας. Για αυτόν ακριβώς το λόγο, η πληροφορία σχετικά με την ολική κατάσταση φθοράς της ομάδας θα πρέπει να είναι όσο το δυνατόν ακριβής γίνεται.

Μία απλή προσέγγιση είναι να χρησιμοποιήσουμε ως πληροφορία της περίληψης, τον μέσο αριθμό σβησίματος της ομάδας (average erase cycle). Κάτι τέτοιο είναι εύκολα υλοποιήσιμο και ο μέσος όρος επαναυπολογίζεται κάθε φορά που ένα μπλοκ της ομάδας υφίσταται σβήσιμο. Για παράδειγμα, εάν ένας μετρητής σβησίματος ανανεώνεται από ec σε ec', τότε ο μέσος αριθμός σβησίματος της ομάδας avg, ισούται με $(avg \times N + ec' - ec)/N$, όπου με *N* συμβολίζουμε τον αριθμό των μπλοκ της ομάδας. Παρ'όλα αυτά, η παραπάνω απλή προσέγγιση δεν είναι αρκετά ακριβής έτσι ώστε να εκφράσουμε την ολική κατάσταση φθοράς μίας ομάδας. Πιο συγκεκριμένα, υποθέτουμε ότι αρκετά μπλοκ μίας ομάδας ανταλλάσσονται με παλαιά μπλοκ. Σε αυτή τη περίπτωση, ο μέσος αριθμός σβησίματος της ομάδας αυξάνεται πολύ περισσότερο σε σύγκριση με τον μετρητή σβησίματος των νέων μπλοκ της ομάδας, με αποτέλεσμα αυτά τα νέα μπλοκ να μην επιλέγονται στη συνέχεια για ανταλλαγή.

Για την βελτίωση της ακρίβειας της πληροφορίας που περιέχεται στην περίληψη μίας ομάδας, προτείνεται η χρήση δύο μέσων τιμών σε κάθε ομάδα. Η μία ονομάζεται **ολική μέση τιμή (total average)** και συμβολίζεται με AVG_T, ενώ η άλλη ονομάζεται **μερική μέση τιμή (partial average)** και συμβολίζεται με AVG_P. Η ολική μέση τιμή καθορίζει το μέσο αριθμό σβησίματος όλων των μπλοκ μίας ομάδας, ενώ η μερική μέση τιμή το μέσο αριθμό σβησίματος των μπλοκ που δεν έχουν υποστεί ακόμη ανταλλαγή. Αρχικά, τόσο η AVG_T όσο και η AVG_P έχουν την ίδια τιμή. Όταν ένα νεό μπλοκ ανταλλάσσεται με ένα παλαιό μπλοκ ανανέωσης, οι τιμές των AVG_T και AVG_P, επαναυπολογίζονται ως εξής:

$$AVG_T = AVG_T + (EC_{old} - EC_{young})/N \quad (1)$$

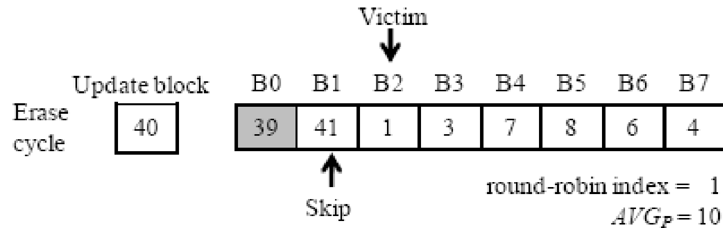
$$AVG_P = ((AVG_P \times n) - EC_{young})/(n-1) \quad (2)$$

Οι τιμές EC_{young} και EC_{old}, αντιπροσωπεύουν τον μετρητή σβησίματος του νέου μπλοκ και του παλαιού μπλοκ ανανέωσης, αντίστοιχα. Με *N* συμβολίζουμε τον αριθμό των μπλοκ σε μία ομάδα, ενώ με *n* τον αριθμό των μπλοκ που δεν έχουν υποστεί ακόμα ανταλλαγή. Εάν όλα τα μπλοκ μίας ομάδας έχουν υποστεί ανταλλαγή, οι τιμές AVG_P και *n* αρχικοποιούνται σε AVG_T και 0, αντίστοιχα.

Για να ελεγχθεί το κατά πόσο χρειάζεται ανταλλαγή «καυτών» - «παγωμένων» δεδομένων, εξετάζεται η διαφορά μεταξύ της τιμής AVG_P και του μετρητή σβησίματος του μπλοκ ανανέωσης, σε σχέση με την τιμή κατωφλίου *TH*. Εάν είναι απαραίτητη η ανταλλαγή «καυτών» - «παγωμένων» δεδομένων, ο αλγόριθμος επιλέγει τη νεότερη ομάδα που έχει την μικρότερη τιμή AVG_P. Για την επιλογή του κατάλληλου μπλοκ μίας ομάδας, χρησιμοποιείται ένας **μηχανισμός εκ περιτροπής (round robin mechanism)**. Πιο συγκεκριμένα, χρησιμοποιείται ένας δείκτης εκ περιτροπής (round robin index) σε κάθε ομάδα, ο οποίος δείχνει στο τελευταίο μπλοκ της ομάδας που έχει επιλεγεί για ανταλλαγή. Ο δείκτης εκ περιτροπής διευκολύνει την αναγνώριση του κατά πόσο ένα μπλοκ έχει υποστεί ανταλλαγή ή όχι. Επιπρόσθετα, ο δείκτης εκ περιτροπής μπορεί να αναλάβει το ρόλο του *n* στην σχέση 2.

7.2.3 Αποτρέποντας την εσφαλμένη ανταλλαγή

Παρόλο που η χρήση της τιμής AVG_P είναι αποτελεσματική κατά την επιλογή ομάδας θύματος (victim group), μπορεί να πραγματοποιηθούν ανεπιθύμητες ανταλλαγές μπλοκ (εσφαλμένες ανταλλαγές), μιας και δεν διτηρούνται στη μνήμη οι μετρητές σβησίματος κάθε μπλοκ ξεχωριστά. Το σχήμα 7.1 που ακολουθεί, παρουσιάζει μία περίπτωση κατά την οποία λαμβάνει χώρα μία εσφαλμένη ανταλλαγή.



Σχήμα 7.1: Αποφυγή εσφαλμένης ανταλλαγής.

Υποθέτουμε ότι το κατώφλι *TH* ισούται με 30 και ότι η ομάδα του σχήματος 7.1 επιλέγεται ως θύμα για ανταλλαγή. Το μπλοκ B1, το οποίο υποδεικνύεται από τον δείκτη εκ περιτροπής (round robin index), επιλέγεται ως θύμα για ανταλλαγή με το μπλοκ ανανέωσης. Η συγκεκριμένη ανταλλαγή, έχει αρνητικό αντίκτυπο στην εξισορρόπηση φθοράς της μνήμης φλας, επειδή ο μετρητής σβησίματος του μπλοκ B1 (ο οποίος ισούται με 41) είναι μεγαλύτερος από αυτόν του μπλοκ ανανέωσης (ο οποίος ισούται με 40).

Η εσφαλμένη ανταλλαγή, όχι μόνο μειώνει την αποδοτικότητα του μηχανισμού εξισορρόπησης φθοράς, αλλά παράγει επιπλέον φόρτο στο σύστημα. Επιπλέον, καθώς το μέγεθος της ομάδας μεγαλώνει, η πιθανότητα εσφαλμένης ανταλλαγής αυξάνεται, επειδή ένας μικρός αριθμός μπλοκ με μεγάλους αριθμούς σβησίματος βρίσκονται στην ομάδα.

Για αποτροπή της εσφαλμένης ανταλλαγής, ο αλγόριθμος δεν στηρίζεται αποκλειστικά στην τιμή AVG_P, αλλά λαμβάνεται υπόψιν ο μετρητής σβησίματος κάθε μπλοκ της ομάδας. Εάν η διαφορά ανάμεσα στον μετρητή σβησίματος του μπλοκ ανανέωσης και του μετρητή σβησίματος του μπλοκ στόχου/θύματος, είναι μικρότερη από $(1-\lambda)\times TH$, το συγκεκριμένο μπλοκ θύμα παρακάμπτεται, και εξετάζεται το αμέσως επόμενο μπλοκ της ομάδας. Το λ είναι μία σταθερά που κυμαίνεται μεταξύ 0 και 1. Η παραπάνω διαδικασία συνεχίζεται μέχρι να συναντηθεί ένα μπλοκ της ομάδας θύματος που να ικανοποιεί την παραπάνω συνθήκη.

Για την υλοποίηση της παραπάνω διαδικασίας, είναι απαραίτητος ένας μηχανισμός για την ανάκτηση του μετρητή σβησίματος ενός μπλοκ. Για την αντιμετώπιση του παραπάνω προβλήματος, αποθηκεύεται ο μετρητής σβησίματος ενός μπλοκ στην **ελεύθερη περιοχή (spare area)**, και ανακτάται κατά τη διάρκεια της λειτουργίας εξισορρόπησης φθοράς. Πιο συγκεκριμένα, αποθηκεύεται ένας μετρητής των τριών byte στην ελεύθερη περοχή της πρώτης σελίδας κάθε μπλοκ. Ο μετρητής ανανεώνεται κάθε φορά που το μπλοκ σβήνεται. Από τη στιγμή που η

ελεύθερη περιοχή μπορεί να γραφτεί ταυτόχρονα με την περιοχή δεδομένων (data area) χρησιμοποιώντας μία μόνο λειτουργία εγγραφής, ο φόρτος που οφείλεται στην ανανέωση του μετρητή είναι αρκετά μικρός. Πιο συγκεκριμένα, ο φόρτος αυτός είναι αμελητέος συγκρινόμενος με το κόστος της εσφαλμένης ανταλλαγής. Ανακεφαλαιώνοντας, ο αλγόριθμος εξισορρόπησης φθοράς που βασίζεται στην ομαδοποίηση, περιλαμβάνει τα παρακάτω βήματα:

1. Όταν το FTL αναθέτει ένα νέο μπλοκ ανανέωσης, τότε επιλέγεται ως θύμα η νεότερη ομάδα η οποία έχει την ελάχιστη τιμή AVG_P.

2. Για τον έλεγχο της συνθήκης εξισορρόπησης φθοράς (wear leveling condition), συγκρίνεται ο μετρητής σβησίματος του μπλοκ ανανέωσης με την τιμή AVG_P της ομάδας θύματος.

3. Εάν η διαφορά είναι μεγαλύτερη του κατωφλίου *TH*, τότε διαβάζεται από την ελεύθερη περιοχή του μπλοκ ο μετρητής σβησίματος του μπλοκ στον οποίο δείχνει ο εκ περιτροπής δείκτης. Ο μετρητής σβησίματος στη συνέχεια, συγκρίνεται με αυτόν του μπλοκ ανανέωσης.

4. Εάν η διαφορά είναι μεγαλύτερη από $(1-\lambda)\times TH$, τότε το μπλοκ επιλέγεται ως θύμα. Διαφορετικά, το μπλοκ παρακάμπτεται και ο αλγόριθμος επιστρέφει στα βήματα 3-4.

5. Τα έγκυρα δεδομένα (ή αλλιώς live data) του μπλοκ θύματος αντιγράφονται στο μπλοκ ανανέωσης, και ο πίνακας μετάφρασης διευθύνσεων τροποποιείται έτσι ώστε να δείχνει προς το μπλοκ ανανέωσης.

6. Στη συνέχεια, το μπλοκ θύμα σβήνεται και κηρύσσεται ως μπλοκ ανανέωσης.

7. Τελικά, οι τιμές AVG_T και AVG_P και ο δείκτης εκ περιτροπής, επαναυπολογίζονται. Εάν ο δείκτης εκ περιτροπής υποδείξει το τελευταίο μπλοκ της ομάδας, η τιμή AVG_P τίθεται ίση με AVG_T, και ο δείκτης εκ περιτροπής ίσος με 0.

7.2.4 Καθορίζοντας τις τιμές των *TH* και *λ*

Στον αλγόριθμο βασιζόμενο στην ομαδοποίηση, το κατώφλι *TH* ελέγχει τον βαθμό της εξισορρόπησης φθοράς. Αυτό οφείλεται στο ότι η απόφαση για το κατά πόσο πρέπει να λάβει χώρα η ανταλλαγή «καυτών» με «παγωμένα» δεδομένα, εξαρτάται από την τιμή του *TH*. Καθώς το κατώφλι *TH* αυξάνεται, ο αλγόριθμος πραγματοποιεί λιγότερες ανταλλαγές και η απόκλιση (deviation) των μετρητών σβησίματος αυξάνεται. Αντιθέτως, μία μικρή τιμή κατωφλίου *TH* έχει ως αποτέλεσμα την αύξηση των λειτουργιών ανταλλαγής και την μείωση της απόκλισης. Παρόλο που μία μικρή τιμή κατωφλίου *TH*, κατανέμει ομοιόμορφα τους μετρητές σβησίματος κατά μήκος της μνήμης φλας, δεν είναι πάντα η πιο σωστή επιλογή μιας και μία πολύ μικρή τιμή *TH* προκαλεί πολλές λειτουργίες ανταλλαγής. Σε αυτή τη περίπτωση, ο μέσος

αριθμός σβησίματος των μπλοκ μίας ομάδας, αυξάνεται πολύ γρήγορα και ο φόρτος που οφείλεται στον μηχανισμό εξισορρόπησης φθοράς γίνεται αρκετά σημαντικός.

Το λ είναι μία σταθερά που κυμαίνεται μεταξύ 0 και 1 και ρυθμίζει το βαθμό αποτροπής της εσφαλμένης ανταλλαγής μπλοκ. Εάν το λ ισούται με 1, ο αλγόριθμος πραγματοποιεί ανταλλαγές μπλοκ, οποτεδήποτε ο μετρητής σβησίματος του μπλοκ της ομάδας θύματος είναι μικρότερος από αυτόν του μπλοκ ανανέωσης. Από την άλλη πλευρά, όταν το λ ισούται με 0, ο μετρητής σβησίματος του μπλοκ πρέπει να είναι μικρότερος από αυτόν του μπλοκ ανανέωσης κατά τουλάχιστον *TH*. Έτσι, όσο μικρότερη είναι η τιμή του λ, τόσο ο αλγόριθμος ψάχνει για το νεότερο μπλοκ μέσα σε μία ομάδα θύμα.

7.2.5 Ανάλυση χωρικής πολυπλοκότητας

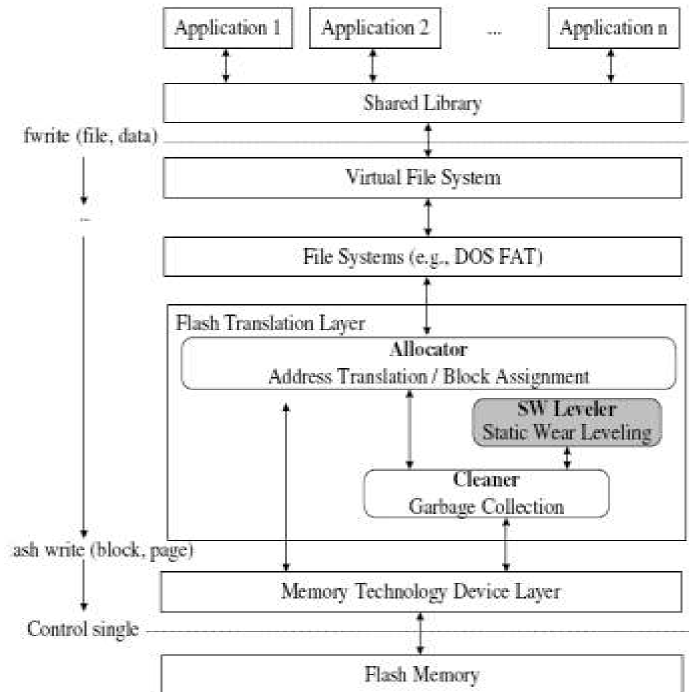
Η αποθήκευση των μετρητών σβησίματος όλων των μπλοκ απαιτεί αρκετό αποθηκευτικό χώρο. Χρειαζόμαστε μετρητές των 20 bits τουλάχιστον, για να αναπαραστίσουμε 1.000.000 σβησίματα. Ο αλγόριθμος που βασίζεται στην ομαδοποίηση, απαιτεί δύο μέσες τιμές των 24 bits για κάθε ομάδα (AVG_T, AVG_P). Επιπρόσθετα, ο δείκτης εκ περιτροπής κάθε ομάδας απαιτεί ένα byte. Ως αποτέλεσμα των παραπάνω, κάθε ομάδα απαιτεί 7 bytes. Εάν για παράδειγμα, μία ομάδα περιλαμβάνει 128 μπλοκ, τότε σε μία 64 Gbyte μνήμη φλας με 4096 ομάδες, το συνολικό μέγεθος της περίληψης πληροφορίας της ομάδας είναι 28 Kbytes.

7.3 Αποδοτικός μηχανισμός στατικής εξισορρόπησης φθοράς

Στην ενότητα αυτή θα παρουσιάσουμε έναν **στατικό μηχανισμό εξισορρόπησης φθοράς (static wear leveling mechanism)**, για την βελτίωση της ανθεκτικότητας (endurance) της μνήμης φλας, με χαμηλές απαιτήσεις σε μνήμη και αρκετά αποδοτική υλοποίηση, χωρίς να χρειάζονται τροποποιήσεις σε επίπεδο FTL ή NFTL.

7.3.1 Επισκόπηση μηχανισμού

Το βασικό κίνητρο της στατικής εξισορρόπησης φθοράς είναι να αποτρέψει τα «παγωμένα» δεδομένα (δεδομένα που ανανεώνονται πολύ σπάνια) από το να παραμένουν σε κάποια μπλοκ για μεγάλα χρονικά διαστήματα. Προσπαθεί να ελαχιστοποιήσει τη μέγιστη διαφορά των μετρητών σβησίματος (erase counters) οποιονδήποτε δύο μπλοκ, έτσι ώστε να αυξηθεί η διάρκεια ζωής της μνήμης φλας. Στο σχήμα 7.2 που ακολουθεί, παρουσιάζεται ο στατικός μηχανισμός εξισορρόπησης φθοράς, ο οποίος ονομάζεται ***SW Leveler*** και υλοποιείται στο στρώμα του FTL. Ο μηχανισμός SW leveler, ενεργοποιεί το **συστατικό εκκαθάρισης (cleaner)** του FTL, έτσι ώστε να εκκινήσει τη διαδικασία συλλογής απορριμμάτων πάνω στα επιλεγμένα μπλοκ και να επιτευχθεί η στατική εξισορρόπηση φθοράς.



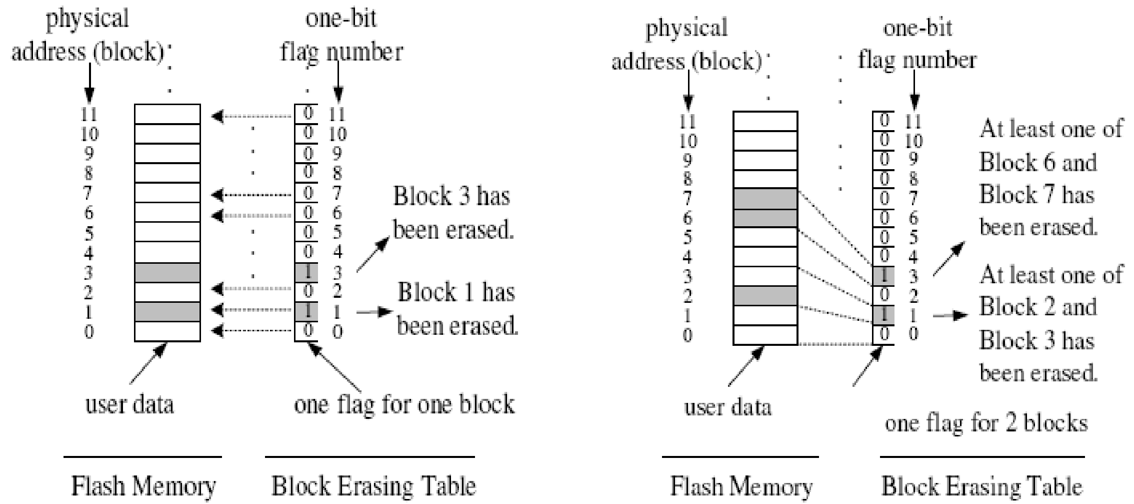
Σχήμα 7.2: Ενσωμάτωση του SW Leveler στο FTL.

Ο μηχανισμός SW Leveler συσχετίζεται με έναν ***πίνακα σβησμένων μπλοκ (Block Erasing Table - BET)***, για να θυμάται ποιό μπλοκ έχει σβηστεί σε μία δεδομένη χρονική στιγμή. Όταν ο μηχανισμός SW Leveler είναι ενεργοποιημένος, τότε είτε επαναπροσδιορίζει (reset) τον πίνακα BET, είτε επιλέγει ένα μπλοκ (με βάση την πληροφορία του πίνακα BET) το οποίο δεν έχει υποστεί ακόμη σβήσιμο, και προκαλεί το συστατικό εκκαθάρισης να εκκινήσει τη διαδικασία συλλογής απορριμμάτων πάνω στο επιλεγμένο μπλοκ. Η διαδιακασία επιλογής του μπλοκ, πρέπει να γίνεται με αποδοτικό τρόπο μέσα σε ένα συγκεκριμένο χρονικό διάστημα. Ο πίνακας BET πρέπει να ανανεώνεται κάθε φορά που ένα μπλοκ υφίσταται σβήσιμο. Όταν ένα μπλοκ της μνήμης φλας ανακυκλώνεται (recycled) μέσω του μηχανισμού συλλογής απορριμμάτων, τότε τροποποιείται ανάλογα και ο απευθείας πίνακας αντιστοίχισης μπλοκ του FTL.

7.3.2 Πίνακας σβησμένων μπλοκ (BET)

Ο σκοπός του πίνακα σβησμένων μπλοκ (BET), είναι να παρέχει πληροφορία για το ποιά μπλοκ έχουν σβηστεί μέσα σε ένα προαποφασισμένο (pre-determined) χρονικό πλαίσιο, γνωστό και ως *διάστημα επαναπροσδιορισμού (resetting interval)*, έτσι ώστε να εντοπίζει μπλοκ που περιέχουν «παγωμένα» δεδομένα. Ο BET είναι ένας πίνακας από bit, στον οποίο κάθε bit συσχετίζεται με ένα σύνολο από 2^k συνεχόμενα μπλοκ. Ο κ είναι ένας ακέραιος ο οποίος είναι μεγαλύτερος ή ίσος του 0. Όταν το συστατικό εκκαθάρισης σβήνει ένα μπλοκ, τότε ο μηχανισμός SW Leveler θέτει το αντίστοιχο bit του πίνακα BET ίσο με 1. Αρχικά, ο πίνακας BET έχει σε όλες τις θέσεις του μηδενικά. Όπως φαίνεται και στο σχήμα 7.3 που ακολουθεί, υπάρχουν δύο διαφορετικοί τρόποι διατήρησης της πληροφορίας σε έναν πίνακα BET. Ο πρώτος τρόπος ονομάζεται ‘ένας προς ένας ’ (One-to-One mode), και ο δεύτερος ‘ένας προς

πολλούς' (One-to-Many mode). Μία «σημαία» (flag), χρησιμοποιείται σε κάθε θέση του πίνακα, για να υποδεικνύει το κατά πόσο κάποια από τα 2^k μπλοκ έχουν σβηστεί.



(α) One-to-One mode. (β) One-to-Many mode

Σχήμα 7.3: (α), (β). Δύο τρόποι διατήρησης της πληροφορίας σε έναν πίνακα BET.

Όταν $\kappa=0$, τότε χρησιμοποιείται μία σημαία για ένα μπλοκ (One-to-One mode). Όσο μεγαλύτερη είναι η τιμή του κ τόσο αυξάνεται η πιθανότητα παράβλεψης μπλοκ με «παγωμένα» δεδομένα. Από την άλλη πλευρά, μία μεγάλη τιμή για το κ βοηθά στο να μειωθούν οι απαιτήσεις σε μνήμη RAM του πίνακα BET.

7.3.3 SW Leveler

Ο μηχανισμός SW Leveler, αποτελείται από τον πίνακα BET καθώς και από δύο διαδικασίες για την πραγματοποίηση της στατικής εξισορρόπησης φθοράς: SWL-Procedure και SWL-BETUpdate. Η διαδικασία SWL-BETUpdate καλείται από το συστατικό εκκαθάρισης του FTL, για να ανανεώσει τον πίνακα BET, μετά από ένα σβήσιμο σε κάποιο μπλοκ της μνήμης φλας. Η διαδικασία SWL-Procedure καλείται όταν χρειάζεται να πραγματοποιηθεί στατική εξισορρόπηση φθοράς. Κάτι τέτοιο γίνεται με την βοήθεια των μεταβλητών f_{cnt} και e_{cnt}, όπου η μεταβλητή f_{cnt} καθορίζει το πλήθος των '1' στον πίνακα BET, ενώ η μεταβλητή e_{cnt} καθορίζει το πλήθος των σβησμένων μπλοκ από τη στιγμή που ο πίνακας BET επαναπροσδιορίστηκε (reset). Όταν ο λόγος e_{cnt} / f_{cnt} (αποκαλείται *unevenness level*) είναι μεγαλύτερος ή ίσος ενός κατωφλίου *T*, τότε καλείται η διαδικασία SWL-Procedure για να ενεργοποιήσει το συστατικό εκκαθάρισης (cleaner) του FTL, έτσι ώστε να εκκινήσει τη διαδικασία συλλογής απορριμμάτων πάνω στα επιλεγμένα μπλοκ και να μετακινηθούν τα «παγωμένα» δεδομένα.

Algorithm 1: SWL-Procedure

```
Input: e_cnt, f_cnt, k, f_index, BET, and T
Output: null
1  if f_cnt = 0 then return ;
2  while e_cnt/f_cnt ≥ T do
       /* size(BET) is the number of flags in the BET.          */
3      if f_cnt ≥ size(BET) then
4          e_cnt ← 0 ;
5          f_cnt ← 0 ;
6          f_index ← RANDOM(0, size(BET) − 1) ;
7          reset all flags in the BET;
8          return;
9      while BET[f_index] = 1 do
10         f_index ← (f_index + 1) mod size(BET)
11     EraseBlockSet(f_index, k) ;   /* Request the Cleaner to do
       garbage collection over the selected block set. */
12     f_index ← (f_index + 1) mod size(BET) ;
```

Ο αλγόριθμος 1 παρουσιάζει τον ψευδικώδικα της διαδικασίας SWL-Procedure. Η διαδικασία επιστρέφει εάν ο πίνακας BET έχει μόλις επαναπροσδιοριστεί (βήμα 1). Όσο ο λόγος e_{cnt} / f_{cnt} είναι μεγαλύτερος ή ίσος του κατωφλίου *T*, το συστατικό εκκαθάρισης ενεργοποιείται σε κάθε επανάληψη, για να πραγματοποιήσει συλλογή απορριμάτων πάνω στο επιλεγμένο σύνολο από μπλοκ (βήματα 2-12). Σε κάθε επανάληψη, ελέγχεται το κατά πόσο όλες οι σημαίες έχουν τιμή ίση με ‘1’. Εάν κάτι τέτοιο ισχύει, ο πίνακας BET επαναπροσδιορίζεται (e_{cnt}=0, f_{cnt}=0). Η μεταβλητή f_{index} υποδεικνύει το σύνολο από μπλοκ για συλλογή απορριμμάτων, και λαμβάνει τυχαία τιμή (βήματα 4-7). Μετά τον επαναπροσδιορισμό του πίνακα BET, η διαδικασία SWL-Procedure επιστρέφει, για να ξεκινήσει το επόμενο διάστημα επαναπροσδιορισμού (resetting interval) (βήμα 8). Διαφορετικά, ο δείκτης f_{index} προχωρά στο επόμενο σύνολο από μπλοκ με μηδενική σημαία (βήματα 9-10). Στη συνέχεια, η διαδικασία SWL-Procedure, ενεργοποιεί το συστατικό εκκαθάρισης (cleaner) του FTL, έτσι ώστε να εκκινήσει τη διαδικασία συλλογής απορριμμάτων πάνω στο επιλεγμένο σύνολο από μπλοκ (βήμα 11), και προχωρά στο επόμενο σύνολο από μπλοκ (βήμα 12). Στο σημείο αυτό πρέπει να αναφέρουμε ότι η μεταβλητή f_{cnt} και ο πίνακας BET ανανεώνονται από τη διαδικασία SWL-BETUpdate. Η επανάληψη της διαδικασίας στατικής εξισορρόπησης φθοράς τερματίζει όταν ο λόγος e_{cnt} / f_{cnt} πέσει σε μία ικανοποιητική τιμή.

Algorithm 2: SWL-BETUpdate

Input: e_{cnt}, f_{cnt}, k, b_{index}, and BET
Output: e_{cnt}, f_{cnt} and BET are updated based on the erased block address b_{index} and k in the BET mapping.

1 $e_{cnt} \leftarrow e_{cnt} + 1$; /* Increase the total erase count. */
/* Update the BET if needed. */
2 **if** $BET[\lfloor b_{index}/2^k \rfloor] = 0$ **then**
3 $\quad BET[\lfloor b_{index}/2^k \rfloor] \leftarrow 1$;
4 $\quad f_{cnt} \leftarrow f_{cnt} + 1$;

Η διαδικασία SWL-BETUpdate παρουσιάζεται στον αλγόριθμο 2. Δοθείσης της διεύθυνσης b_{index} του μπλοκ που σβήστηκε από το συστατικό εκκαθάρισης, η διαδικασία SWL-BETUpdate αυξάνει πρώτα το πλήθος των σβησμένων μπλοκ κατά το διάστημα επαναπροσδιορισμού (βήμα 1). Εάν η αντίστοιχη καταχώριση του πίνακα BET δεν έχει τιμή ίση με 1, τότε η καταχώριση γίνεται ίση με 1, και το πλήθος των ‘1’ στον πίνακα BET αυξάνεται κατά 1 (βήματα 2-4).

7.3.4 Ανάλυση χωρικής πολυπλοκότητας

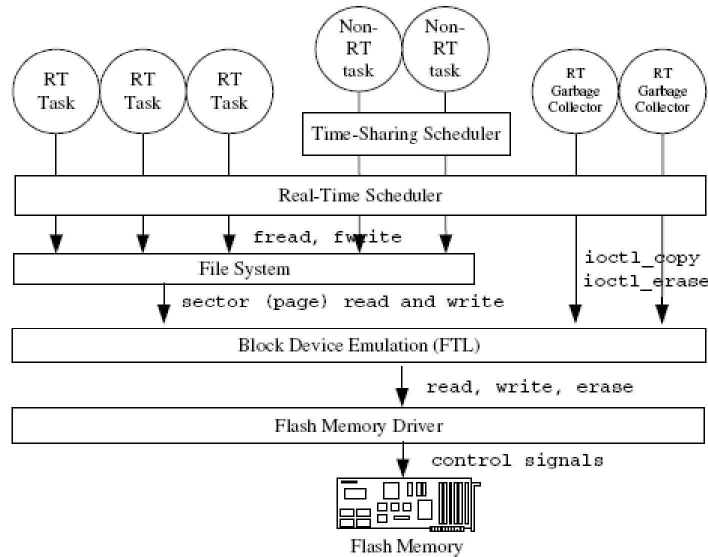
Από τη στιγμή που χρειάζεται μία σημαία του ενός bit για κάθε σύνολο από μπλοκ, ο πίνακας BET προκαλεί τον κύριο φόρτο κύριας μνήμης, για την διατήρηση της κατάστασης σβησίματος των μπλοκ. Όπως φαίνεται και στον πίνακα 7.1, το μέγεθος του πίνακα BET ποικίλλει, ανάλογα με το μέγεθος της μνήμης φλας και την τιμή του *κ*. Για παράδειγμα, το μέγεθος του πίνακα BET είναι ίσο με 512B σε μία SLC μνήμη φλας των 4GB, με *κ*=3. Αν χρησιμοποιήσουμε MLC μνήμη φλας, τότε το μέγεθος του πίνακα BET μικραίνει αρκετά.

	128MB	256MB	512MB	1GB	2GB	4GB
$k = 0$	128B	256B	512B	1024B	2048B	4096B
$k = 1$	64B	128B	256B	512B	1024B	2048B
$k = 2$	32B	64B	128B	256B	512B	1024B
$k = 3$	16B	32B	64B	128B	256B	512B

Πίνακας 7.1: Μέγεθος του πίνακα BET για SLC μνήμη φλας.

7.4 Μηχανισμός συλλογής απορριμμάτων πραγματικού χρόνου (Real-time garbage collection mechanism)

Στις ενότητες που ακολουθούν, θα παρουσιάσουμε ένα μηχανισμό συλλογής απορριμμάτων πραγματικού χρόνου (Real-time garbage collection mechanism), ο οποίος εξασφαλίζει ντετερμινιστική απόδοση σε αποθηκευτικά συστήματα μνήμης φλας πραγματικού χρόνου (real-time systems). Στο σχήμα 7.4 παρουσιάζεται η αρχιτεκτονική ενός αποθηκευτικού συστήματος μνήμης φλας πραγματικού χρόνου.



Σχήμα 7.4: Αρχιτεκτονική συστήματος.

Το σύστημα υποστηρίζει τόσο **εργασίες πραγματικού χρόνου (real-time tasks)**, όσο και **εργασίες μη πραγματικού χρόνου (non-real-time tasks)**, μέσω ενός χρονοπρογραμματιστή πραγματικού χρόνου (real-time scheduler) και ενός χρονοπρογραμματιστή διαμοιραζόμενου χρόνου (time-sharing scheduler), αντίστοιχα. Σε κάθε εργασία πραγματικού χρόνου που γράφει δεδομένα στη μνήμη φλας, ανατίθεται ένας **συλλέκτης απορριμμάτων πραγματικού χρόνου (real-time garbage collector)**, για την ανάκτηση ελεύθερων σελίδων προοριζόμενες για την αντίστοιχη εργασία. Ο συλλέκτης απορριμμάτων πραγματικού χρόνου, αλληλεπιδρά απευθείας με το FTL μέσω ειδικά σχεδιαμένων υπηρεσιών ελέγχου (control services). Στις υπηρεσίες ελέγχου συμπεριλαμβάνονται η *ioctl_erase,* η οποία πραγματοποιεί το σβήσιμο των μπλοκ (block erasing), καθώς και η *ioctl_copy* η οποία πραγματοποιεί την **ατομική αντιγραφή (atomic copying)**. Κάθε ατομική αντιγραφή, η οποία αποτελείται από μία ανάγνωση σελίδας και στη συνέχεια από μία εγγραφή σε επίπεδο σελίδας, έχει ως σκοπό να πραγματοποιεί αντιγραφή σελίδων που περιέχουν πρόσφατα/νέα δεδομένα (live-page copying) κατά τη συλλογή απορριμμάτων. Στο σημείο αυτό, πρέπει να αναφέρουμε ότι δεν δημιουργούνται συλλέκτες απορριμμάτων στην περίπτωση εργασιών μη πραγματικού χρόνου. Αντιθέτως, το FTL αναλαμβάνει την ανάκτηση ελεύθερου αποθηκευτικού χώρου για τις εργασίες μη πραγματικού χρόνου.

7.4.1 Μοντέλο εργασιών πραγματικού χρόνου

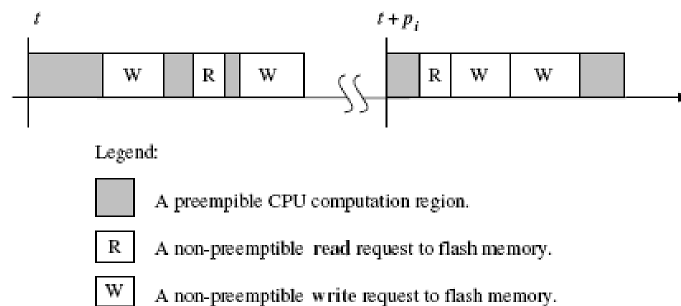
Όπως είναι γνωστό και από το πρώτο κεφάλαιο, ένα τσιπάκι μνήμης φλας μπορεί να υποστηρίξει τριών ειδών πράξεις: ανάγνωση σελίδας (page read), εγγραφή σε επίπεδο σελίδας (page write) και σβήσιμο σε επίπεδο μπλοκ (block erase). Οι επιδόσεις των παραπάνω πράξεων επιδεικνύονται στον πίνακα που ακολουθεί.

	Page Read 512 bytes	Page Write 512 bytes	Block Erase 16K bytes
Performance(μs)	348	909	1,881
Symbol	t_r	t_w	t_e

Πίνακας 7.2: Επιδόσεις NAND μνήμης φλας.

Όπως εύκολα μπορεί να παρατηρήσει κάποιος, η πράξη του σβησίματος μπλοκ διαρκεί πολύ περισσότερο χρόνο σε σύγκριση με τις άλλες δύο λειτουργίες. Οι παραπάνω πράξεις καταναλώνουν πολλούς κύκλους μηχανής της CPU. Αυτό έχει ως αποτέλεσμα, η CPU να είναι πλήρως απασχολημένη κατά την διάρκεια της εκτέλεσης μίας εκ των τριών πράξεων.

Στην ενότητα αυτή, θα αναλύσουμε τις απαιτήσεις μίας εργασίας πραγματικού χρόνου, τόσο σε επίπεδο CPU όσο και σε επίπεδο υπηρεσιών αποθηκευτικού συστήματος μνήμης φλας. Πιο συγκεκριμένα, κάθε περιοδική εργασία πραγματικού χρόνου T_i ορίζεται ως μία τριάδα (c_{T_i}, p_{T_i}, w_{T_i}). Με c_{T_i} συμβολίζουμε τις απαιτήσεις σε CPU, με p_{T_i} την περίοδο και με w_{T_i} το μέγιστο αριθμό εγγραφών σε επίπεδο σελίδας ανά περίοδο, της εργασίας πραγματικού χρόνου αντίστοιχα. Οι απαιτήσεις c_{T_i} σε CPU περιλαμβάνουν, το χρόνο χρήσης της CPU για υπολογισμούς (CPU computation time) καθώς και το χρόνο εκτέλεσης λειτουργιών στη μνήμη φλας (flash memory operating time). Υποθέτουμε ότι η εργασία T_i επιθυμεί να χρησιμοποιήσει τη CPU για $c_{T_i}^{cpu}$ μs, διαβάζει *i* σελίδες από τη μνήμη φλας και γράφει *j* σελίδες στη μνήμη φλας, ανά περίοδο. Οι απαιτήσεις c_{T_i} σε CPU υπολογίζονται ως εξής: $c_{T_i}^{cpu}$ + $i*t_w + j*t_r$ (τα σύμβολα t_w και t_r βρίσκονται στον πίνακα 7.2). Εάν η εργασία T_i δεν πραγματοποιεί εγγραφή σελίδας, τότε θέτουμε w_{T_i}=0. Στο σχήμα 7.5 που ακολουθεί, παρουσιάζεται μία περιοδική εργασία πραγματικού χρόνου T_i, η οποία πραγματοποιεί μία ανάγνωση σελίδας και δύο εγγραφές σε επίπεδο σελίδας, σε κάθε περίοδο.



Σχήμα 7.5: Μία εργασία T_i που διαβάζει και γράφει τη μνήμη φλας.

Στις ενότητες που ακολουθούν θα παρουσιαστεί ο μηχανισμός συλλογής απορριμμάτων πραγματικού χρόνου. Αρχικά, θα παρουσιατεί η ιδέα των συλλεκτών απορριμμάτων πραγματικού χρόνου (real-time garbage collectors), εν συνεχεία ο μηχανισμός ανανέωσης ελεύθερων σελίδων (free-page replenishment mechanism),

και στο τέλος η πολιτική ανακύκλωσης μπλοκ (block-recycling policy) η οποία επιλέγει τα κατάλληλα μπλοκ προς ανακύκλωση.

7.4.2 Συλλέκτες απορριμμάτων πραγματικού χρόνου

Για κάθε εργασία πραγματικού χρόνου T_i, η οποία μπορεί να γράψει δεδομένα στη μνήμη φλας (w_{T_i}>0), δημιουργείται ένας αντίστοιχος **συλλέκτης απορριμμάτων πραγματικού χρόνου (real-time garbage collector)** G_i. Ο συλλέκτης G_i ανακυκλώνει ένα μπλοκ σε κάθε περίοδο, ενώ παράλληλα ανακτά και παρέχει ελεύθερες σελίδες στην εργασία T_i. Έστω ότι μία σταθερά α, αναπαριστά το κατώτερο όριο στον αριθμό ελεύθερων σελίδων που ανακτώνται κατά την ανακύκλωση ενός μπλοκ. Έστω ότι η σταθερά π αναπαριστά τον αριθμό σελίδων ανά μπλοκ. Οι παραπάνω συμβολισμοί, καθώς και κάποιοι άλλοι που θα χρησιμοποιηθούν στις μετέπειτα ενότητες, παρουσιάζονται συγκεντρωτικά στον πίνακα 7.3 που ακολουθεί.

symbol	Description
Λ	the total number of live pages currently on flash
Δ	the total number of dead pages currently on flash
Φ	the total number of free pages currently on flash
π	the number of pages in each block
Θ	the total number of pages on flash (=$\Lambda + \Delta + \Phi$)
α	the constant lower-bound of the number of reclaimed free pages after each block recycling
ρ	the total number of tokens in system
ρ_{free}	the number of unallocated tokens in system
ρ_{T_i}	the number of tokens given to T_i
ρ_{G_i}	the number of tokens given to G_i
ρ_{nr}	the number of tokens given to non-real-time tasks

Πίνακας 7.3: Ορισμοί συμβόλων.

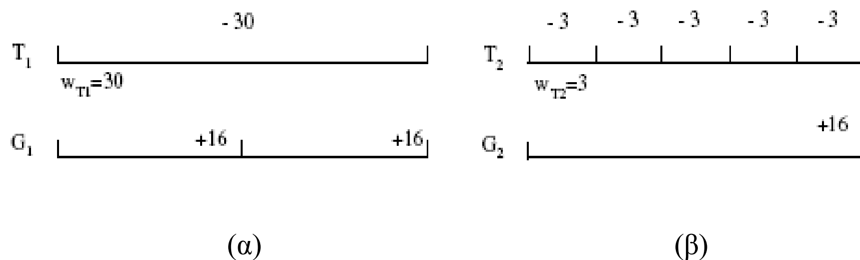
Δοθείσης μίας εργασίας πραγματικού χρόνου $T_i = (c_{T_i},\ p_{T_i},\ w_{T_i})$ με w_{T_i}>0, ο αντίστοιχος συλλέκτης απορριμμάτων πραγματικού χρόνου δημιουργείται ως εξής:

$$c_{G_i} = (\pi - \alpha) * (t_r + t_w) + t_e + c_{G_i}^{cpu}$$

(1)

$$p_{G_i} = \begin{cases} p_{T_i} / \left\lceil \frac{w_{T_i}}{\alpha} \right\rceil, if w_{T_i} > \alpha \\ p_{T_i} * \left\lfloor \frac{\alpha}{w_{T_i}} \right\rfloor, otherwise \end{cases}$$

Οι απαιτήσεις c_{G_i} σε CPU, περιλαμβάνουν το πολύ $(\pi-\alpha)$ αντιγραφές «ζωντανών» σελίδων (live-page copyings), ένα σβήσιμο σε μπλοκ (block erase) καθώς και τις απαιτήσεις υπολογισμού $c_{G_i}^{cpu}$. Η χειρότερη περίπτωση των απιτήσεων σε CPU, είναι ίδια για όλους τους συλλέκτες απορριμμάτων πραγματικού χρόνου, μιάς και η παράμετρος α είναι ένα σταθερό κατώτερο όριο. Προφανώς, η παραπάνω σχέση παρέχει μία εκτίμηση των απαιτήσεων σε CPU, και ο συλλέκτης G_i είναι πιθανό να μην τις καταναλώσει εξολοκλήρου σε μία περίοδο. Η περίοδος p_{G_i} εξαρτάται από το πόσο γρήγορα καταναλώνει τις ελέυθερες σελίδες η εργασία T_i. Υποθέτουμε ότι τόσο ο συλλέκτης G_i όσο και η εργασία T_i, καταφθάνουν στο σύστημα την ίδια ακριβώς χρονική στιγμή.



(α) (β)

Σχήμα 7.6: Δημιουργία συλλεκτών απορριμμάτων πραγματικού χρόνου.

Το σχήμα 7.6 παρουσιάζει δύο παραδείγματα συλλεκτών απορριμμάτων πραγματικού χρόνου, με $\alpha=16$. Στο σχήμα 7.6 (α), επειδή w_{T_1}=30>16, η περίοδος p_{G_1} ισούται με το μισό της περιόδου p_{T_1} (δηλ. p_{G_1}= p_{T_1}/2), έτσι ώστε ο συλλέκτης G_1 να ανακτήσει 32 σελίδες για την εργασία T_1 σε μία περίοδο p_{T_1}. Στο σχήμα 7.6 (β), επειδή w_{T_2}=3<16, η περίοδος p_{G_2} είναι ίση με p_{G_2}=5* p_{T_2}, έτσι ώστε ο συλλέκτης G_2 να ανακτήσει 16 σελίδες για την εργασία T_2 σε μία περίοδο p_{G_2}. Οι ανακτόμενες σελίδες καλύπτουν τις ανάγκες των εργασιών T_1 και T_2 αντίστοιχα. Ορίζουμε ως **μεταπερίοδο (meta-period)** σ_i μίας εργασίας πραγματικού χρόνου T_i και ενός συλλέκτη G_i, την τιμή $\sigma_i=p_{T_i}$, εάν $p_{T_i}\geq p_{G_i}$, διαφορετικά $\sigma_i=p_{G_i}$. Στο παραπάνω παράδειγμα, σ_1 = p_{T_1} και σ_2 = p_{G_2}.

7.4.3 Μηχανισμός ανανέωσης ελεύθερων σελίδων

Στην ενότητα αυτή, θα παρουσιάσουμε ένα **μηχανισμό ανανέωσης ελεύθερων σελίδων βασισμένο σε αναγνωριστικά (token-based free page replenishment mechanism)**, για την αποδοτική και ευέλικτη διαχείριση των ελεύθερων σελίδων καθώς και της ανάκτησής τους.

Έστω μία εργασία πραγματικού χρόνου T_i= (c_{T_i}, p_{T_i}, w_{T_i}) με w_{T_i}>0. Αρχικά, η εργασία T_i λαμβάνει ($w_{T_i}*\sigma_i/p_{T_i}$) αναγνωριστικά (tokens), ενώ ένα αναγνωριστικό είναι αρκετό για την εκτέλεση μίας εγγραφής σε επίπεδο σελίδας (page write). Υποθέτουμε ότι μία εργασία πραγματικού χρόνου δεν μπορεί να γράψει καμία σελίδα

εάν δεν έχει στην κατοχή της κάποιο αναγνωριστικό. Ένα αναγνωριστικό, δεν αντιστοιχεί σε κάποια συγκεκριμένη ελεύθερη σελίδα στο σύστημα, ενώ παράλληλα χρησιμοποιούνται πολλοί μετρητές αναγνωριστικών (πίνακας 7.3). Πιο συγκεκριμένα, με ρ_{init} συμβολίζουμε το συνολικό αριθμό διαθέσιμων αναγνωριστικών κατά την εκκίνηση του συστήματος, με ρ συμβολίζουμε το συνολικό αριθμό αναγνωριστικών που βρίσκονται ήδη στο σύστημα (ανεξάρτητα από το αν έχουν ανατεθεί ή όχι), με ρ_{free} συμβολίζουμε το πλήθος των αναγνωριστικών που δεν έχουν ακόμη ανατεθεί σε κάποια εργασία, ενώ με ρ_{T_i} και ρ_{G_i} συμβολίζουμε τον αριθμό των αναγνωριστικών που έχουν ανατεθεί στην εργασία T_i και τον συλλέκτη G_i, αντίστοιχα.

Αρχικά, η εργασία T_i λαμβάνει ($w_{T_i} * \sigma_i / p_{T_i}$) αναγνωριστικά, ενώ ο συλλέκτης G_i, ($\pi - \alpha$) αναγνωριστικά αντίστοιχα. Κάτι τέτοιο έχει ως σκοπό να αποφύγουμε το μπλοκάρισμα των T_i και G_i στην πρώτη τους μεταπερίοδο. Κατά τη διάρκεια της εκτέλεσης, ο συλλέκτης G_i και η εργασία T_i συμπεριφέρονται ως ένα ζευγάρι παραγωγού-καταναλωτή αναγνωριστικών, αντίστοιχα. Πιο συγκεκριμένα, ο συλλέκτης G_i δημιουργεί και παρέχει αναγνωριστικά στην εργασία T_i, σε κάθε περίοδο p_{G_i}, εξαιτίας των ανακτόμενων ελεύθερων σελίδων κατά την ανακύκλωση μπλοκ. Η ανανέωση των αναγνωριστικών είναι αρκετή έτσι ώστε η εργασία T_i να πραγματοποιεί εγγραφές σελίδων σε κάθε μεταπερίοδό της σ_i. Όταν η εργασία T_i καταναλώνει ένα αναγνωριστικό, τόσο ο αριθμός ρ_{T_i} όσο και ο ρ, μειώνονται κατά μία μονάδα. Αντίστοιχα, όταν ο συλλέκτης ανακτά μία ελεύθερη σελίδα, με αποτέλεσμα να δημιουργείται ένα αναγνωριστικό, τότε τόσο ο αριθμός ρ_{G_i} όσο και ο ρ, αυξάνονται κατά μία μονάδα. Στο τέλος κάθε περιόδου p_{G_i}, ο συλλέκτης G_i παρέχει στην εργασία T_i τα αναγνωριστικά που έχει δημιουργήσει. Όταν η εργασία T_i και ο συλλέκτης G_i τερματιστούν, τότε τα αναγνωριστικά τους πρέπει να επιστραφούν στο σύστημα.

Ο παραπάνω μηχανισμός ανανέωσης έχει ορισμένα προβλήματα: Αρχικά, οι εργασίες πραγματικού χρόνου είναι πιθανό να καταναλώνουν τις ελεύθερες σελίδες με μικρότερο ρυθμό σε σχέση με αυτόν που είχαν δηλώσει. Δεύτερον, είναι πιθανό οι συλλέκτες απορριμμάτων πραγματικού χρόνου να ανακτούν πολύ περισσότερες σελίδες σε σχέση με αυτές που είχαμε εκτιμήσει στη χειρότερη περίπτωση. Από τη στιγμή που ο συλλέκτης G_i ανεφοδιάζει την εργασία T_i με τουλάχιστον α αναγνωριστικά, τότε αυτά τα αναγνωριστικά θα συσσωρεύονται βαθμιαία στην εργασία T_i. Ένα άλλο πρόβλημα είναι το ότι ο συλλέκτης G_i ανακτά άσκοπα ελεύθερες σελίδες, ακόμη και αν υπάρχει ικανοποιητικός αριθμός ελεύθερων σελίδων στο σύστημα. Ένας βελτιωμένος μηχανισμός ανανέωσης, παρουσιάζεται παρακάτω:

Στην αρχή κάθε μεταπεριόδου σ_i, η εργασία T_i παραδίδει ρ_{T_i} - ($w_{T_i} * \sigma_i / p_{T_i}$) αναγνωριστικά, μιας και αυτά τα αναγνωριστικά είναι πέραν των αναγκών της, και μειώνει τους αριθμούς ρ_{T_i} και ρ με τον παραπάνω όρο. Στην αρχή κάθε περιόδου

p_{G_i}, ο συλλέκτης απορριμμάτων G_i ελέγχει το κατά πόσο ισχύει η ανισότητα $(\Phi - \rho) \geq \alpha$, όπου με Φ συμβολίζουμε τον αριθμό ελεύθερων σελίδων που είναι διαθέσιμες στο σύστημα. Εάν η συνθήκη ισχύει, τότε ο συλλέκτης G_i παίρνει α ελεύθερες σελίδες από το σύστημα, και οι αριθμοί ρ_{G_i} και ρ αυξάνονται κατά α, αντί ουσιατικά να γίνει μία ανακύκλωση μπλοκ. Σε διαφορετική περίπτωση (δηλ. $(\Phi - \rho) < \alpha$), ο συλλέκτης G_i πραγματοποιεί ανακύκλωση μπλοκ για την ανάκτηση ελεύθερων σελίδων και την δημιουργία αναγνωριστικών. Υποθέτουμε ότι ο συλλέκτης G_i έχει στην κατοχή του y αναγνωριστικά, και ότι στη συνέχεια δίνει στην εργασία T_i α αναγνωριστικά. Ο συλλέκτης G_i είναι πιθανό να παραδόσει $y - \alpha - (\pi - \alpha) = y - \pi$ αναγνωριστικά, μιας και είναι πέραν των αναγκών της. Παρακάτω, παρουσιάζεται ο αλγόριθμος του μηχανισμού ανανέωσης ελεύθερων σελίδων.

```
Ti()
{
    if(beginningOfMetaPeriod())
    {
      // give up the extra tokens
      x=ρTi-(wTi*σi)/pTi;
      ρTi=ρTi-x;ρ=ρ-x;
    }
    ....... /* job of Ti */
}
```

```
Gi()
{
    if((Φ-ρ)≥α)
    {
        // create tokens from the
        // existing free pages
        ρGi=ρGi+α;ρ=ρ+α;
    }
    else
    {
        // Recycle a block
        recycleBlock();
        // Note that Φ,ρ,ρGi change
    }
    // Supply Ti with α tokens
     ρTi=ρTi+α;ρGi=ρGi-α;

    // Give up the residual tokens
     z=ρGi-(π-α);ρGi=ρGi-z;ρ=ρ-z;

}
```

7.4.4 Πολιτική ανακύκλωσης μπλοκ

Η **πολιτική ανακύκλωσης μπλοκ (block-recycling policy)** αποφασίζει για το ποιό μπλοκ πρέπει να σβηστεί κατά τη διάρκεια της συλλογής απορριμμάτων. Στην ενότητα αυτή, θα παρουσιάσουμε μία «άπληστη» (greedy) πολιτική, η οποία υπόσχεται μία προβλέψιμη απόδοση κατά την συλλογή απορριμμάτων.

Πιο συγκεκριμένα, προτείνεται η χρήση μίας πολιτικής που ανακυκλώνει το μπλοκ που περιέχει τις περισσότερες «νεκρές» (dead) σελίδες. Προφανώς, η χειρότερη περίπτωση όσον αφορά το πλήθος των ανακτόμενων ελεύθερων σελίδων, εμφανίζεται όταν οι «νεκρές» σελίδες ισοκατανέμονται σε όλα τα μπλοκ του συστήματος. Ο αριθμός των ελεύθερων σελίδων που ανακτώνται στη χειρότερη περίπτωση, αμέσως μετά την ανακύκλωση ενός μπλοκ, δίνεται από τον παρακάτω τύπο:

$$\left\lceil \pi * \frac{\Delta}{\Theta} \right\rceil \qquad (2)$$

όπου με π, Δ και Θ συμβολίζουμε τον αριθμό σελίδων ανά μπλοκ, το συνολικό αριθμό «νεκρών» σελίδων στη μνήμη φλας και τον συνολικό αριθμό σελίδων στη μνήμη φλας, αντίστοιχα. Η παραπάνω σχέση καθορίζει την απόδοση χειρότερης περίπτωσης της «άπληστης» πολιτικής, η οποία είναι ανάλογη του λόγου του πλήθους «νεκρών» σελίδων προς το πλήθος όλων των σελίδων της μνήμης φλας. Αυτή είνα μία πάρα πολύ σημαντική παρατήρηση, μιας και διαπιστώνουμε ότι δεν εξασφαλίζουμε στον μηχανισμό συλλογής απορριμμάτων υψηλή απόδοση, με το να αυξάνουμε την χωρητικότητα της μνήμης φλας. Οι παράμετροι π και Θ είναι σταθερές του συστήματος. Μία «άπληστη» πολιτική που θα διαχειρίζεται κατάλληλα την παράμετρο Δ, θα έχει ως αποτέλεσμα την βελτίωση του κατώτερου ορίου όσον αφορά το πλήθος των ελεύθερων σελίδων που ανακτώνται κατά την ανακύκλωση ενός μπλοκ.

Επειδή ισχύει $\Theta = \Delta + \Phi + \Lambda$, η σχέση 2 μπορεί να γραφτεί ως εξής:

$$\left\lceil \pi * \frac{\Theta - \Phi - \Lambda}{\Theta} \right\rceil \quad (3)$$

όπου με Φ και Λ συμβολίζουμε το πλήθος των ελεύθερων σελίδων και το πλήθος των «ζωντανών» (live) σελίδων (πίνακας 7.3), αντίστοιχα, όταν είναι ενεργοποιημένη η πολιτική ανακύκλωσης μπλοκ. Όπως φαίνεται και από τη σχέση 3, η απόδοση χειρότερης περίπτωσης της πολιτικής ανακύκλωσης μπλοκ ελέγχεται φράσσοντας τις τιμές των Φ και Λ. Η πολιτική ανακύκλωσης μπλοκ ενεργοποιείται μόνο όταν ο αριθμός των ελεύθερων σελίδων είναι μικρότερος του φράγματος για το Φ.

7.4.5 Υποστήριξη εργασιών μη πραγματικού χρόνου

Στην ενότητα αυτή, θα επεκτείνουμε το μηχανισμό ανανέωσης ελεύθερων σελίδων βασισμένο σε αναγνωριστικά, έτσι ώστε να παρέχει ελεύθερες σελίδες και σε εργασίες μη πραγματικού χρόνου (non-real-time tasks). Στη συνέχεια θα παρουσιαστεί ένας εξισορροπιστής φθοράς μη πραγματικού χρόνου (non-real time wear leveller), για την βελτίωση της ανθεκτικότητας της μνήμης φλας.

7.4.5.1 Ανανέωση ελεύθερων σελίδων για εργασίες μη πραγματικού χρόνου

Αντίθετα από τις εργασίες πραγματικού χρόνου, οι εργασίες μη πραγματικού χρόνου μοιράζονται μεταξύ τους μία συλλογή αναγνωριστικών ρ_{nr}. Έστω ότι δίνονται αρχικά π αναγνωριστικά στις εργασίες μη πραγματικού χρόνου ($\rho_{nr} = \pi$). Προτού μία εργασία μη πραγματικού χρόνου πραγματοποιήσει μία εγγραφή σελίδας, πρέπει να ελέγξει το κατά πόσο ισχύει η συνθήκη $\rho_{nr} > \pi$. Εάν η συνθήκη ισχύει, τότε η εγγραφή σελίδας πραγματοποιείται, ενώ παράλληλα καταναλώνεται και ένα αναγνωριστικό (οι αριθμοί ρ_{nr} και ρ μειώνονται κατά μία μονάδα). Σε διαφορετική περίπτωση, το σύστημα πρέπει να εφοδιαστεί με αναγνωριστικά τα οποία θα προορίζονται για εργασίες μη πραγματικού χρόνου. Η δημιουργία αναγνωριστικών για εργασίες μη πραγματικού χρόνου, ακολουθεί παρόμοια στρατηγική με αυτή των

συλλεκτών απορριμμάτων πραγματικού χρόνου: Εάν $\Phi \le \rho$, τότε πραγματοποιείται ανακύκλωση ενός μπλοκ για την ανάκτηση ελεύθερων σελίδων και τη δημιουργία αναγνωριστικών. Εάν $\Phi > \rho$, τότε πιθανότατα δεν υπάρχει ανάγκη για ανακύκλωση μπλοκ.

7.4.5.2 Εξισορροπιστής φθοράς μη πραγματικού χρόνου

Στον υπό εξέταση μηχανισμό της ενότητας 7.4, προτείνεται η χρήση μίας εργασίας μη πραγματικού χρόνου η οποία θα πραγματοποιεί την εξισορρόπηση φθοράς. Επίσης, διαχωρίζεται η πολιτική εξισορρόπηση φθοράς από την πολιτική ανακύκλωσης των μπλοκ. Πιο συγκεκριμένα, προτείνεται η κατασκευή ενός **εξισορροπιστή φθοράς μη πραγματικού χρόνου (non-real time wear-leveler)**, ο οποίος σαρώνει σειριακά έναν αριθμό από μπλοκ για να διαπιστώσει το κατά πόσο κάποιο από αυτά τα μπλοκ έχουν σχετικά μικρό αριθμό σβησίματος (small erase count). Ένας από τους βασικότερους λόγους για τους οποίους ένα μπλοκ έχει σχετικά μικρό αριθμό/πλήθος σβησίματος, είναι το γεγονός ότι περιέχει πολλά «παγωμένα» δεδομένα (δεδομένα που ανανεώνονται σπανίως). Τέτοιου είδους μπλοκ δεν είναι καλοί υποψήφιοι για ανακύκλωση, με αποτέλεσμα να σβήνονται πολύ σπάνια. Όταν ο εξισορροπιστής φθοράς ανακαλύπτει ένα μπλοκ με σχετικά μικρό αριθμό σβησίματος, τότε ο εξισορροπιστής αντιγράφει σε πρώτη φάση τις «ζωντανές» σελίδες του μπλοκ σε κάποιο άλλο μπλοκ. Καθώς ο εξισορροπιστής φθοράς επαναλαμβάνει τη σάρωση και την αντιγραφή «ζωντανών» σελίδων, οι «νεκρές» σελίδες του μπλοκ στόχου, θα αυξάνονται βαθμιαία. Σαν αποτέλεσμα των παραπάνω, αργά η γρήγορα αυτά τα μπλοκ θα επιλεγούν από την πολιτική ανακύκλωσης των μπλοκ. Η βασική ιδέα είναι να «ξεπαγώσουμε» τα μπλοκ μετακινόντας τα «παγωμένα» δεδομένα.

Βιβλιογραφία κεφαλαίου

[1] Dawoon Jung, Yoon-Hee Chae, Heeseung Jo, Jin-Soo Kim and Joonwon Lee. A Group-Based Wear-Leveling Algorithm for Large-Capacity Flash Memory Storage Systems. (χρήση στις σελίδες 113-118)

[2] Yuan-Hao Chang, Jen-Wei Hsieh and Tei-Wei Kuo. Endurance Enhancement of Flash-Memory Storage Systems: An Efficient Static Wear Levelling Design. (χρήση στις σελίδες 118-122)

[3] Li-Pin Chang and Tei-Wei Kuo. A Real-Time Garbage Collection Mechanism for Flash-Memory Storage Systems in Embedded Systems. (χρήση στις σελίδες 122-130)

Συνολική βιβλιογραφία

[1] Eran Gal and Sivan Toledo. Algorithms and Data Structures for Flash Memories. (χρήση στις σελίδες 8-9, 17-20, 21-22, 27-28, 31-33, 36-38,39-40, 41-43)

[2] NAND vs. NOR Flash Memory. Technology Overview, Toshiba America

Electronic Components. (χρήση στις σελίδες 11-14)

[3] Siddhart Choudhuri and Tony Givargis. Performance Improvement of Block Based NAND Flash Translation Layer. (χρήση στις σελίδες 23-27)

[4] David WoodHouse. JFFS: The Journalling Flash File System. Red Hat, Inc. (χρήση στις σελίδες 34-38)

[5] www.yaffs.net. (χρήση στις σελίδες 38, 41)

[6] Chin-Hsien Wu, Tei-Wei Kuo and Li Ping Chang. An Efficient B-Tree Layer Implementation for Flash-Memory Storage Systems. (χρήση στις σελίδες 47-64)

[7] Dongwon Kang, Dawoon Jung, Jeong-Uk Kang and Jin-Soo Kim. μ- Tree: An Ordered Index Structure for NAND Flash Memory. (χρήση στις σελίδες 9-10, 64-78)

[8] Suman Nath and Aman Kansal. FlashDB: Dynamic Self-tuning Database for NAND Flash. (χρήση στις σελίδες 78-87)

[9] A. Guttman. R-tree: A dynamic index structure for spatial searching, 1984. (χρήση στις σελίδες 90-98)

[10] Chin-Hsien Wu, Li-Oin Chang, Tei-Wei Kuo. An efficient R-tree implementation over flash memory storage systems 2003. (χρήση στις σελίδες 98-104)

[11] Jen-Wei Hsieh, Li-Pin Chang, Tei-Wei Kuo. Efficient On-line Identification of Hot Data for Flash-Memory Management. (χρήση στις σελίδες 107-110)

[12] Dawoon Jung, Yoon-Hee Chae, Heeseung Jo, Jin-Soo Kim and Joonwon Lee. A Group-Based Wear-Leveling Algorithm for Large-Capacity Flash Memory Storage Systems. (χρήση στις σελίδες 113-118)

[13] Yuan-Hao Chang, Jen-Wei Hsieh and Tei-Wei Kuo. Endurance Enhancement of Flash-Memory Storage Systems: An Efficient Static Wear Levelling Design. (χρήση στις σελίδες 20, 23, 118-122)

[14] Li-Pin Chang and Tei-Wei Kuo. A Real-Time Garbage Collection Mechanism for Flash-Memory Storage Systems in Embedded Systems. (χρήση στις σελιδες 122-130)

[15] www.wikipedia.org (χρήση στις σελίδες 12-13)

[16] www.google.com